SAND99-0000
Unlimited Release
Printed June 1999

Distribution
Category UC-705

DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Sensitivity Analysis, and Uncertainty Quantification

Michael S. Eldred
Structural Dynamics Department

William J. Bohnhoff
Simulation Technology Research Department

William E. Hart
Applied Mathematics Department

Sandia National Laboratories
Albuquerque, New Mexico 87185

Abstract

The DAKOTA (Design Analysis Kit for OpTimizAtion) iterator toolkit is a flexible, extensible interface between simulation codes and iterative systems analysis methods. The toolkit implements optimization with a variety of gradient and nongradient-based methods, uncertainty quantification with nondeterministic propagation methods, parameter estimation with nonlinear least squares solution methods, and sensitivity analysis with general-purpose parameter study capabilities. By employing object-oriented design to implement abstractions of the key concepts involved in iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment which uses point solutions from simulation codes for answering fundamental engineering questions, such as “what is the best design?”, “how safe is it?”, or “how much confidence do I have in my answer?”.

In addition to these iterative systems analysis capabilities, advanced users can employ state of the art capabilities for (1) exploiting parallelism at multiple levels (coarse-grained and fine-grained) and (2) building cascaded, nested, concurrent, and/or adaptive strategies which utilize multiple iterators and models to enact hybridization, sequential approximation, stochastic optimization, or mixed continuous-discrete optimization.

This report serves as a user’s guide and reference for the DAKOTA software and provides capability overviews, command specifications, and installation and execution instructions.

Acknowledgment

Development of DAKOTA was funded by the Engineering Science Research Foundation, the Computer Science Research Foundation, Laboratory Directed Research and Development, the Surety Defense Programs backbone, and the Accelerated Strategic Computing Initiative.

The development of this software involved many technical staff and contractors across Sandia-Albuquerque and Sandia-Livermore. The authors would like to recognize Chris Moen, Roy Lee, Juan Meza, and Charles Tong for their support of the OPT++ library and the SGI platform; Michael Brayer, Craig Shierling, and Brian Driessen for their help in implementing the IDR parser within DAKOTA; Jim Schutt for his consultation on IDR capabilities; Bruce Bainbridge and Vicente Romero for their work on DAKOTA's nondeterministic methods; Todd Simmermacher for his work on the multidimensional parameter study method and the continuation method for failure capturing; Brian Dennis and Chris O'Gorman for their work on the approximation interface class hierarchy; Ron Rhea for his work on the configuration system; and David Outka for his vision and numerous contributions during the initial stages of the DAKOTA development effort.

The authors also greatly appreciate the helpful comments made by Ben Blackwell and Todd Simmermacher during the review process.

Documentation Versions

Since the DAKOTA architecture is flexible and extensible, its capabilities are continuously evolving. Therefore, the DAKOTA software documentation is a living document for which this SAND report is one snapshot in time. Updated documentation will be provided with major software releases.

The software documentation can be published using either hardcopy or online formats. The hardcopy format is used for generating the SAND report version, whereas the online format is used for Web publishing of a hyperlinked Portable Document Format (PDF) version. At the time of printing, the online PDF document is publicly available from

http://endo.sandia.gov/DAKOTA/papers/Dakota_online.pdf

Contact the authors at mseldre@sandia.gov if problems are encountered in accessing this file.

Table of Contents

List of Figures

List of Tables

DAKOTA Introduction

Motivation

Advanced computational methods have been developed for simulating complex physical systems in disciplines such as fluid mechanics, structural dynamics, heat transfer, nonlinear structural mechanics, shock physics, and many others. In many situations simulators can be used to generate highly accurate models of real processes. These simulators can be an enormous aid to engineers who want to develop an understanding and/or predictive capability for the complex behaviors that are often observed in the respective physical systems. Often, these simulators are employed as virtual prototypes, where a set of predefined system parameters, such as size or location dimensions and material properties, are adjusted to improve or optimize the performance of a particular system, as defined by one or more system performance objectives. Optimization of the virtual prototype then requires running the simulator, evaluation the performance objective(s), and adjusting the system parameters in an iterative and directed way, such that an improved or optimal solution is obtained for the simulation as measured by the performance objective(s). System performance objectives can be formulated, for example, to minimize weight, cost, or defects; to limit a critical temperature, stress, or vibration response; or to maximize performance, reliability, throughput, reconfigurability, agility, or design robustness. One of the primary motivations for the development of DAKOTA has been to provide engineers with a systematic and rapid means of obtaining improved or optimal design approximations from their simulator-based models. Making this capability available to engineers generally leads to better designs and improved system performance at earlier stages of the design phase, and eliminates some of the dependence on real prototypes and testing, thereby shortening the design cycle and reducing overall product development costs.

In addition to improving performance objectives through optimization, computational simulations can also be used as tools to quantify uncertainty and assess risk in high-consequence events, to investigate the sensitivity of critical responses to model variations, and to reconcile model predictions with experimental observations. In each of these studies (as well as many others), computational simulations are used to provide the necessary informational building blocks for answering fundamental engineering questions about the predictive accuracy of computational models and the performance, safety, and reliability of products and processes. By providing a flexible and extensible framework for the answering of these fundamental questions, the utility and impact of computational methods can be greatly extended. This is what the DAKOTA activity strives to achieve.

What is DAKOTA?

The DAKOTA (Design Analysis Kit for OpTimizAtion) provides a flexible, extensible interface between your simulator and a variety of iterative methods and strategies. While DAKOTA was

originally conceived as an easy-to-use interface between simulation codes and numerical optimization codes, recent versions have been expanded to include other types of iterative analysis. In addition to an abundance of optimization methods and strategies that it supports, the present version of DAKOTA also implements uncertainty quantification with nondeterministic propagation methods, parameter estimation with nonlinear least squares solution methods, and sensitivity analysis with general-purpose parameter study capabilities. Thus, one of the many advantages that DAKOTA has to offer is that access to a very broad range of iterative capabilities can be obtained through a single, relatively simple interface between DAKOTA and your simulator. DAKOTA manages interfacing with the iterative methods and strategies, relieving you of this often difficult and time consuming development burden.

Each of the numerical iterative methods supported by DAKOTA executes your simulation code at a series of different design parameter values. DAKOTA, in conjunction with the iterative methods that it supports, can utilize the this series of point solutions from your simulation code to answer fundamental engineering questions, such as “what is the best design?”, “how safe is it?”, or “how much confidence do I have in my answer?”. In addition to providing this environment for answering systems performance questions, the DAKOTA toolkit also provides an extensible platform for the development of customized methods and strategies, which can be used to increase the robustness and efficiency of the iterative analyses for computationally complex engineering problems (see [Eldred, M.S., 1998]).

The DAKOTA toolkit is a flexible problem-solving environment that offers a systematic way of obtaining iterative solutions to user generated design problems. Should you want to try a different type of iterative method or strategy with your simulator, it will only be necessary to change a relatively few commands in the DAKOTA input and start a new analysis. The flexible yet systematic approach to DAKOTA command syntax allows you to change between methods and strategies in an efficient manner, the need to learn a completely different style of command syntax and the need to reconstruct of the interface each time you want to use a new optimization or other iterator method is eliminated.

Five architectural components define and control the flow of data through DAKOTA, these are: **strategies**, **methods**, **variables**, **responses**, and **interfaces**. These five components define separate areas of flexibility and extensibility. *Strategies* manage the interplay of the other components and allow you to build sophisticated and adaptive schemes based on method combination and hybridization, management of approximate models, incorporation of uncertainty into optimization processes, management of parallelism, etc. Other novel approaches to the systems analysis process can be added as they are envisioned and used to leverage the developments within the other architecture components. *Methods* include the major categories optimization, uncertainty quantification, nonlinear least squares, and parameter study, and are extensible, both through the inclusion of new algorithms within a category, and through the addition of new iterator branches that fit the general model of repeated mapping of variables into responses through simulation codes. *Variables* currently include design, uncertain, and state variable specifications for continuous, discrete, and mixed problem domains. *Responses* include function values, gradients, and Hessians (an optimization data set), where these functions can be

objective and constraint functions, residual functions (least squares data set), or generic response functions (uncertainty and parameter study data sets) depending on the iterator in use. Lastly, *interfaces* provide access to simulation codes, test functions, and approximations through a variety of communication protocols. In the DAKOTA architecture, *strategies* manage how *methods* map *variables* into *responses* through the use of *interfaces*.

Tutorial

Getting started

In this section you will be given instructions on how to set up and run a simple DAKOTA optimization analysis. It is assumed that the DAKOTA install procedure, as outlined in the Installation Guide on page 180, has been completed successfully, including configuration with the NPSOL and/or DOT optimization package(s) enabled. Once DAKOTA has been successfully installed you are ready to proceed with the tutorial. A later tutorial example will show you how to set up and run a DAKOTA analysis in parallel processing mode. If you intend to run this example you will need to configure DAKOTA with MPI as described in Configuring with the Message Passing Interface on page 184.

The getting started tutorial will proceed by having you set up and run a sample numerical optimization problem in DAKOTA. In this tutorial you will learn how to:

- Construct a simple interface between an evaluation code and DAKOTA
- set up a DAKOTA input file including strategy, interface, variables, responses, and method specifications
- initiate a DAKOTA run
- interpret a DAKOTA output file

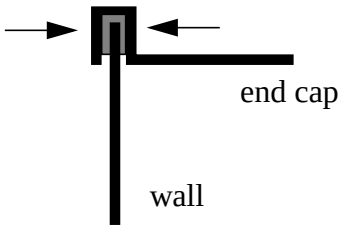
Working through the example should give you a good understanding of the basic operation of DAKOTA. Additional examples, which will allow you to further your understanding of DAKOTA, appear in the sections titled Textbook Example on page 192, Rosenbrock Example on page 204, Cylinder Head Example on page 208, Engineering Applications on page 216, and Some useful features of DAKOTA on page 35, as well as throughout the text.

A basic optimization problem

As a means of familiarizing new users to the DAKOTA software and as a means of demonstrating some of the capabilities of DAKOTA, a simple example optimization problem will be worked. For this example, suppose that a high-volume manufacturer of light weight steel containers wants to minimize the amount of raw sheet material that must be used to manufacture a 1.1 quart cylindrical-shaped can, including waste material. Material for the container walls and end caps is stamped from stock sheet material of constant thickness. The seal between the end

caps and container wall is manufactured by a press forming operation on the end caps. The end caps can then be attached to the container wall forming a seal through a crimping operation.

Figure 1 Container wall-to-end-cap seal.



For preliminary design purposes, the extra material that would normally go into the container end cap seals is approximated by increasing the cut dimensions of the end cap diameters by 12% and the height of the container wall by 5%, and waste associated with stamping the end caps in a specialized pattern from sheet stock is estimated as 15% of the cap area. The equation for the area of the container materials including waste is

$$A = 2\times\left(\begin{array}{c}\texttt{end cap}\\ \texttt{waste}\\ \texttt{material}\\ \texttt{factor}\end{array}\right)\times\left(\begin{array}{c}\texttt{end cap}\\ \texttt{seal}\\ \texttt{material}\\ \texttt{factor}\end{array}\right)\times\left(\begin{array}{c}\texttt{nominal}\\ \texttt{end cap}\\ \texttt{area}\end{array}\right)+\left(\begin{array}{c}\texttt{container}\\ \texttt{wall seal}\\ \texttt{material}\\ \texttt{factor}\end{array}\right)\times\left(\begin{array}{c}\texttt{nomina}\\ \texttt{contain}\\ \texttt{wall are}\end{array}\right) \quad (1)$$

or

$$A = 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)\pi DH \quad (2)$$

where D and H are the diameter and height of the finished product in units of inches, respectively. The volume of the finished product is given by

$$V = \pi\frac{D^2H}{4} = (1.1\texttt{qt})(57.75\texttt{in}^3/\texttt{qt}) \quad (3)$$

The equation for area is the objective function for this problem; it is to be minimized. The equation for volume is an equality constraint; it must be satisfied at the conclusion of the optimization problem. Any combination of D an H that satisfy the volume constraint produce a **feasible** solution (although not necessarily the optimal solution) to the area minimization problem, and any combination that do not satisfy the volume constraint generate an **infeasible** solution. Thus, in this optimization problem, the area objective function is to be minimized with respect to parameters D and H, subject to satisfaction of the volume constraint. The area that is a

minimum subject to the volume constraint is the **optimal** area, and the corresponding values for the parameters D and H are the optimal parameter values. The optimization problem can be stated in a more compact and standardized form as

$$\begin{aligned} \min \quad & 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)\pi DH \\ \text{subject to:} \quad & \pi\frac{D^2 H}{4} - (1.1\,qt)(57.75\,in^3/qt) = 0 \end{aligned} \tag{4}$$

It is important that the equations supplied to a numerical optimization code be limited to generating only physically realizable parameters as optimizers. It is often up to the engineer to supply these limits, usually in the form of parameter bound constraints. General purpose numerical optimizers do not typically have the capability to differentiate between physically meaningful and unmeaningful parameter values. For example, by observing the equations for the area objective function and the volume constraint, it can be seen that by allowing the diameter, D, to become negative, it is algebraically possible to generate relatively small values for the area that also satisfy the volume constraint. Negative values for D are of course physically meaningless. Therefore, to ensure that the numerically-solved optimization problem remains meaningful, a bound constraint of $D \geq 0$ must be included in the optimization problem statement. A positive value for H is implied since the volume constraint could never be satisfied if H were negative. However, a bound constraint of $H \geq 0$ can be added to the optimization problem if desired.

A graphical view of the container optimization problem appears in Figure 2. The 3-D surface defines the area, A, as a function of diameter and height. The curved line that extends across the surface defines the areas that satisfy the volume equality constraint, V. Graphically, the container optimization problem can be viewed as one of finding the point along the constraint line with the smallest 3-D surface height in Figure 2. This point corresponds to the optimal or minimizing values for diameter and height of the final product.

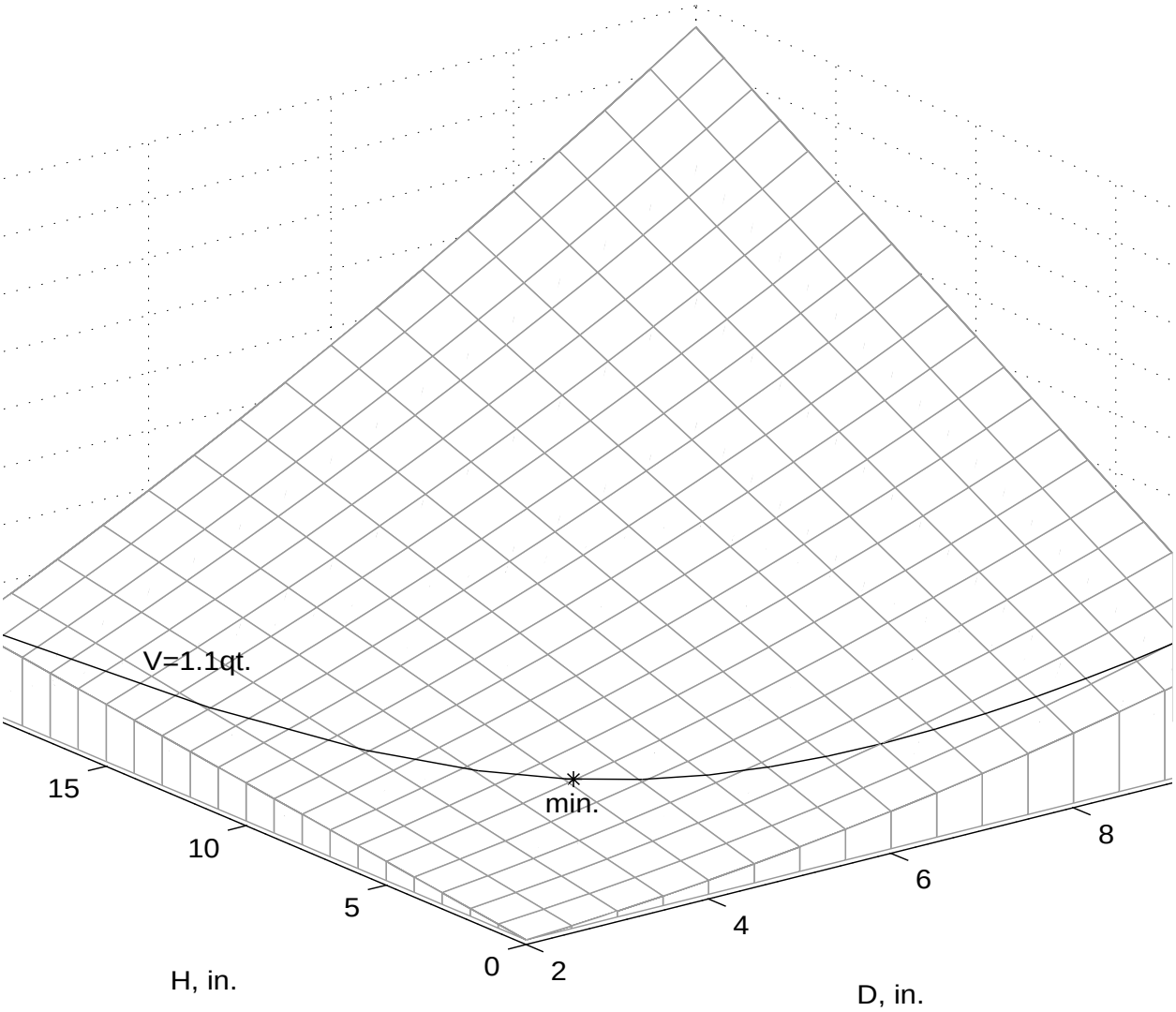


Figure 2 A graphical representation of the container optimization problem.

The numerical optimizers that are presently supported by DAKOTA accept only inequality constraints, in a less-than-or-equal-to format, and not equality constraints such as the volume constraint in this example. However, it is possible to represent any equality constraint, such as $g(x) = 0$, with two inequality constraints, $g(x) \leq 0$ and $-g(x) \leq 0$, since the only time both inequalities are satisfied is when $g(x) = 0$ is satisfied. Given the requirements on the constraint functions and variable bounds, the optimization problem can restated as

$$\begin{aligned} \min \quad & 2(1.15)(1.12)\pi\frac{D^2}{4} + (1.05)^2\pi DH \\ \text{subject to:} \quad & \pi\frac{D^2H}{4} - (1.1qt)(57.75in^3/qt) \le 0 \\ & -\pi\frac{D^2H}{4} + (1.1qt)(57.75in^3/qt) \le 0 \\ & D \ge 0, H \ge 0 \end{aligned} \tag{5}$$

This statement of the optimization problem will be incorporated into a simulator in the following sections. The term **simulator** is defined within DAKOTA in a general sense. A simulator is any computer code that can accept variables as input, and compute and output responses in the form of function values and possibly gradient and second partial derivative (Hessian) information. In terms of the DAKOTA iterator for this optimization example, D and H are **variables,** and the area objective function, and the volume constraint functions are contained within the simulator, and are to be used to generate **responses**. Bound constraints are handled internally by optimizers and do not need to be managed via a users interface. The mechanisms for receiving the variables from DAKOTA into the simulator, computing the responses, and passing the responses from the simulator back to DAKOTA comprise the **interface**. What remains to be done before DAKOTA can be used to solve this optimization problem is the construction of this interface, and selection of one or more **methods** and **strategies** from the DAKOTA library. These tasks will be covered in the following sections.

Constructing the interface

An **interface** in the DAKOTA environment is a user routine that is responsible for mapping variables into responses. While a practical implementation of an interface might include calls to a finite element or finite difference simulation code, a simple example interface will be constructed in this section that will be used to compute values for the area objective function and the volume constraint functions from algebraic equations using values of D and H as input variables. Code for reading the input variables and writing the output responses is also part of the interface.

DAKOTA offers more than one option for initiating execution of the interface and for performing the input of variables and output of responses. For the purpose of an introductory example the The System Call Application Interface on page 81 approach will be used to initiate execution of the interface. Another interface possibility is given in the section titled The Direct Function Application Interface on page 80. For the system call approach, the interface exists as one or more stand-alone executable programs. One execution of the interface reads one set of variables, executes the simulator, which performs any necessary calculations, and outputs one set of responses. For this example the 1-piece Interface on page 81 will be used. For this example the interface will house the input, computational, and output parts of the interface in a single executable. The 3-piece Interface on page 81 is an alternative that can be used to obtain a preprocessor-simulator-postprocessor interface format. Example listings of the interface for the

container optimization problem are given in Figure 3 through Figure 5 for Fortran, C, and C++ languages, respectively.

Figure 3 Fortran listing of the interface for the container example.

```
c************************************************************************
c************************************************************************
      program container
c************************************************************************
c************************************************************************
      integer num_fns,num_vars,req(1:3)
      double precision fval(1:3),D,H
      character*80 infile,outfile,instr
      character*25 valtag(1:3)
      double precision PI /3.14159265358979/

c     get the input and output file names from the command line
c     using the fortran 77 library routine getarg
      call getarg(1,infile)
      call getarg(2,outfile)

c************************************
c     read the input data from DAKOTA
c************************************
      open(11,FILE=infile,STATUS='OLD')

c     get the number of variables and function evaluation requests
      read(11,*)num_vars,instr,num_fns,instr

c     get the values of the variables and the associated tag names
      read(11,*)H,instr
      read(11,*)D,instr

c     get the evaluation type request for the associated function number
      do 10 i=1,num_fns
        read(11,*)req(i),instr
 10   continue

      close(11)


c************************************************************
c     compute the objective function and constraint values
c************************************************************
      if(req(1).eq.1) fval(1)=0.644*PI*D**2+1.05*PI*D*H
      if(req(2).eq.1) fval(2)=0.25*PI*H*D**2-63.525
      if(req(3).eq.1) fval(3)=-0.25*PI*H*D**2+63.525

c***************************************
c     write the response output for DAKOTA
c***************************************
      valtag(1)='area'
      valtag(2)='volume_constraint_1'
      valtag(3)='volume_constraint_2'

      open(11,FILE=outfile,STATUS='UNKNOWN')

      do 20 i=1,num_fns
        if(req(i).eq.1) then
          write(11,'(E22.15,1X,A)'),fval(i),valtag(i)
        endif
 20   continue

      close(11)

      end
```

The one-piece approach assumes that all file I/O pre and post-processing are present in one callable program or driver routine. File names are supplied on the command line for the

interface, e.g. an internal system call by DAKOTA to the one-piece interface looks something like:

```
system("container variables.in responses.out");
```

where `container` is the simulator executable for this example, and the variables input and responses output file names follow on the same line. File names can then be accessed by the interface using a command line argument procedure (library routine `getarg` in Fortran or the array `argv` in C or C++). While not strictly needed when file names are not changing, command line retrieval of the file names is required when unique name assignment (e.g. file tagging) is used.

Figure 4 C language listing of the container simulator example.

```
#include <stdio.h>
#include <stdlib.h>

/*************************************************************************/
/* container.c - container optimization example                          */
/*************************************************************************/
void main(int argc, char **argv)
{
FILE *fileptr;
double fval[3],D,H;
int i,num_vars,num_fns,req[3];
char *infile,*outfile,in_str[81];
char *valtag[]={"area\n",
                "volume_constraint_1\n",
                "volume_constraint_2\n"};
const double PI = 3.14159265358979;

/* assign the input and output file names from the command line */
infile = argv[1];
outfile = argv[2];

/*****************************/
/* read the input from DAKOTA */
/*****************************/
fileptr = fopen(infile,"r");

/* get the number of variables and functions*/
fscanf(fileptr,"%d %80s %d %80s",&num_vars,in_str,&num_fns,in_str);

/* get the values of the variables and the associated tag names */
fscanf(fileptr,"%lf %80s",&H,in_str);
fscanf(fileptr,"%lf %80s",&D,in_str);

/* get the evaluation type request */
for(i=0; i<num_fns; i++)
  fscanf(fileptr,"%d %80s",&req[i],in_str);

fclose(fileptr);

/*******************************************************/
/* compute the objective function and constraint values */
/*******************************************************/
 if(req[0]==1)
    fval[0]=0.644*PI*D*D+1.04*PI*D*H;
 if(req[1]==1)
    fval[1]=0.25*PI*H*D*D-63.525;
 if(req[2]==1)
    fval[2]=-0.25*PI*H*D*D+63.525;

/*************************************/
/* write the response output for DAKOTA */
/*************************************/
fileptr = fopen(outfile,"w");
```

```
for(i=0; i<num_fns; i++)
  if(req[i]!=0)
    fprintf(fileptr,"%23.15e %s",fval[i],valtag[i]);

fclose(fileptr);

}
```

For the one-piece interface, the i/o routines associated with the simulator must be able to read and write files in one of the allowable DAKOTA formats. For the purposes of this example the input file generated by DAKOTA for the simulator will have the following format:

```
2 variables 3 functions
<double> D
<double> H
1 ASV_1
1 ASV_2
1 ASV_3
```

The simulator must be able to read this file to compute the objective and constraint function values. The first line of the file indicates that there are two variables for this optimization problem: `D` and `H`, and three functions: (1) the area objective function and (2) and (3) the volume constraint functions. Bound constraints do not need to be computed by the simulator. The second and third lines are used to transmit values of the variables `D` and `H` from DAKOTA to the simulator. The `<double>` descriptors represent real values of `D` and `H` that would appear in an actual simulator input file. The last three lines are encoded requests for the type of computation that is to be associated with each of the three functions. The value of `1` in first character position of the last three lines indicates that a function value is being requested for each of the three functions. Other numbers can be used to make requests for gradient or Hessian information, or some combination of the function, gradient, and Hessian information, see the section titled DAKOTA File Data Formats on page 85 and specifically the subsection Active set vector control on page 90 for more information. However, for this example, only function values will be requested by DAKOTA and any gradient information needed by the numerical optimizer will be computed internally by DAKOTA through finite differencing. The strings beginning with ASV on the last three lines of the file are the default tag names for each function. Function numbers `1` through `3` in the on the end positions of the last three lines correspond to the functions labeled `fval(1)` through `fval(3)` in the Fortran listing for the container simulator, or functions labeled `fval[0]` through `fval[2]` in the C and C++ listings, respectively. It is also possible to assign tag names to these requests.

Figure 5 C++ listing of the container optimization example

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

//------------------------------------------------------------------------
// container.C - C++ container optimization example
//------------------------------------------------------------------------

int main(int argc, char** argv)
{

  //****************************
  // read the input from DAKOTA
```

```
  //****************************
  fstream fin(argv[1],ios::in);

  // get the number of variables and functions
  int num_vars, num_fns;
  char in_str[81];
  fin >> num_vars >> in_str >> num_fns >> in_str;

  // get the values of the variables and the associated tag names
  double D,H;
  fin >> H >> in_str;
  fin >> D >> in_str;

  // get the evaluation type request
  int* req = new int [num_fns];
  int i;
  for(i=0; i<num_fns; i++) {
    fin >> req[i];
    fin.ignore(256, '\n');
  }

  fin.close();

  //*********************************************************
  // compute the objective function and constraint values
  //*********************************************************
  double *fval = new double [num_fns];
  const double PI = 3.14159265358979;
  if(req[0]==1)
    fval[0]=0.644*PI*D*D+1.04*PI*D*H;
  if(req[1]==1)
    fval[1]=0.25*PI*H*D*D-63.525;
  if(req[2]==1)
    fval[2]=-0.25*PI*H*D*D+63.525;

  //***************************************
  // write the response output for DAKOTA
  //***************************************
  fstream fout(argv[2],ios::out);
  fout.precision(15);
  fout.setf(ios::scientific);
  fout.setf(ios::right);
  char *val_tag[]= {"area\n",
                    "volume_constraint_1\n",
                    "volume_constraint_2\n"};

  for(i=0; i<num_fns; i++)
    if(req[i]=1)
      fout << setw(23) << fval[i] << " " << val_tag[i];

  fout.close();

  return 0;
}
```

In this example `num_fns` represents the total number of objective and/or constraint function evaluations in the model. For the container optimization example there is one area objective function and two volume inequality constraint functions. Requests for a function evaluation are stored in variable `req(i);` the objective function request is stored in `req(1)` and the volume constraint requests are stored in `req(2)` and `req(3)`, respectively, for the Fortran listing. A value of `1` for `req(i)` indicates compute the associated function evaluation, while a value of `0` indicates do not compute. The objective function value is stored in `fval(1)` and the volume constraint values are stored in `fval(2)` and `fval(3)`, respectively. For this example the evaluation request (stored in `req(i)`) will consist strictly of requests or nonrequests for

function values. Any gradient or Hessian information needed by the numerical optimizer is computed internally by DAKOTA through finite differencing and additional calls to the simulator, thereby relieving you of this burden. However, if the interface has the capability to compute gradient and/or Hessian information internally, DAKOTA also has the capability to make requests for this information if it is needed by the numerical optimizer. Such an interface could contain branching and looping structures to handle specific requests for gradient and Hessian information. However, the limited complexity of these versions of the interface are suitable for this simple example.

The simulator-to-DAKOTA response output has the following format for the container optimization problem:

```
<double> area
<double> volume_constraint_pos
<double> volume_constraint_neg
```

This file contains one line for each of the function values that was requested in the simulator input file. The <double> descriptors represent real values of each associated function tag (`area`, for example). The function tags are optional. They are in fact ignored by DAKOTA, and the order of the numeric data is assumed to be in the same as the order of requests in the input file. Function tags do however increase the readability of the output files. The only requirements for function tags is that they be separated from the numeric data by a blank space or new line character, that they contain at least one character (A-Z or a-z), and that they contain no blank spaces. Output of gradient and Hessian information is also possible. See Results file format on page 88 for more information.

Creating a DAKOTA input file

A DAKOTA input file is a collection of character and numeric information that describes the problem to be solved. For this example, the file will be named `dakota_container.in`. The input file contains fields describing what strategy, method, variables, responses, and interface components of DAKOTA are to be used to solve the problem. The contents of the DAKOTA input file must not conflict with the problem as defined in the simulator. A DAKOTA input file for the container optimization problem is given in Figure 6. Any line beginning with a ‘#’ character is treated as a comment. Presence of the backslash (\) character is required in the input file to indicate the continuation of a major specification (`interface`, `variables`, `strategy`, `method`, or `response`) onto the next line of the file. The last line of each specification is not terminated with a ’\’ character since it marks the specification’s end.

Figure 6 DAKOTA input file for the container optimization example.

```
# Interface specification
       interface,                                                    \
         application, system                                         \
         analysis_driver = 'container'

# Variables specification
       variables,                                                    \
         continuous_design = 2                                       \
         cdv_descriptor 'H' 'D'                                      \
         cdv_initial_point 4.5 4.5                                   \
         cdv_lower_bounds 0.0 0.0
```

```
# Strategy specification
       strategy,                                                          \
         single_method

# Method specification
       method,                                                            \
         npsol_sqp

# Responses speification
       responses,                                                         \
         num_objective_functions = 1                                      \
         num_nonlinear_constraints = 2                                    \
         numerical_gradients                                              \
         method_source dakota                                             \
         interval_type central                                            \
         fd_step_size = 0.001                                             \
         no_hessians
```

In the first four lines of `dakota_container.in` the interface specification is made. The system call application interface is specified with the command

```
application, system\
```

and `container`, the name of the executable simulator file, is specified as the analysis driver on the following line.

Next, the strategy and method specifications are made. For this example a `single_method` strategy is specified, which means that only one optimizer will be used to perform the analysis. The numerical optimizer that will be used for this analysis is the `npsol_sqp` optimizer. This optimizer is selected in the method specification. The NPSOL library provides an implementation of the SQP or sequential quadratic programming method for nonlinearly constrained local optimization. For this method it is assumed that the objective and constraint functions have continuous first and second partial derivatives. It is also implied that the problem possesses a single local optimal value. However, this method can be applied to problems with more than one local optimum, if the locally optimal value is considered to be of use even though it may not be the global optimum.

Following specification of the method, the variables specifications are made. For this example, the number of design variables is equal to `2` and this count is set with the command

```
continuous_design = 2\
```

where `continuous_design` variables have been specified since any real value within the bounds is a possible solution. Next the name tags for the optimization variables (`D` and `H`) are set with the command

```
cdv_descriptor 'H' 'D'\
```

where `cdv_descriptor` stands for continuous design variable descriptor. This is followed by the `cdv_initial_point` to be used at the start of the optimization analysis, and then the values of the variable bounds. Since only lower bounds are specified, the problem is unbounded above.

After declaring the variables their associated specifications are, the response specification is made in file `dakota_container.in`. First, the number of objective functions is set to `1` (for the area objective function) and the number of nonlinear constraints (the volume constraints in

this example) is set to `2`. The following four lines in the response specification state that central finite difference gradients are to be used by the numerical optimizer, and that these gradients are to be computed by DAKOTA using a step size of `0.001`. These specifications are necessary since they control what DAKOTA asks for and expects in the simulator input and response output files, respectively, and what internal computations are to be performed on the simulator response output to generate the gradient approximation. The command `no_hessians` is specified since the interface will not return the Hessian information, rather the `npsol_sqp` numerical optimizer generates its own internal gradient-based Hessian approximation.

The `npsol_sqp` optimizer was selected because it has the capability to handle nonlinear objective and constraint functions. The `dot_mmfd`, `dot_slp`, and `dot_sqp` methods also possess these capabilities. DAKOTA can be used to easily change between installed numerical optimizers. For the DOT optimizer methods this can be achieved by simply replacing the method specification `npsol_sqp` in the DAKOTA input file with one of the three appropriate DOT methods. See NPSOL Method on page 162 and DOT Methods on page 161 for additional information.

Running DAKOTA

Once the interface has been constructed, the process of executing DAKOTA for the example problem is relatively simple. One possible way to execute the example is to place `dakota_container.in` and the interface executable, `container`, in a directory with a path to the DAKOTA executable. The directory $DAKOTA/test is one such directory. It is also possible to create a link to the dakota executable with the UNIX `ln` command in some other directory. If the container simulator executable has not been created it will be necessary to do so with a command such as

```
f77 -o container container.f
```

for Fortran, or

```
cc -o container container.c
```

for C, or

```
CC -o container container.C
```

for C++. The actual compile commands may vary from system to system. What is important is that an executable, of one of the preceding example simulators, with the name `container` exists in the working directory for this example. Once the files are located in an appropriate directory DAKOTA is executed from the UNIX prompt for the container example with the command:

```
dakota -i dakota_container.in
```

DAKOTA should take a few seconds to load and execute. Output should print to the standard output device. The DAKOTA output can also be redirected to a file using the syntax

```
dakota -i dakota_container.in > dakota.out
```

where `dakota.out` can be replaced by any desired file name. Output will be discussed in the following section. See Running DAKOTA on page 123 for a more detailed discussion.

Interpreting the results

Figure 7 shows a partial listing of the output for the container optimization example. The first several lines, down to the line that reads "`Running Single Method Strategy...`", reflect information that was specified in the DAKOTA input file or during DAKOTA installation. The lines that follow, down to the line that begins with "`NPSOL exits with INFORM code = 0`", contain information about the function and gradient evaluations that have been requested by NPSOL. Several of the function evaluations and gradient-related function evaluations have been omitted from this listing for brevity.

The values of the optimization variables and the initial objective and constraint function evaluations are listed following the line that reads "`Begin Function Evaluation 1`". The values of the optimization variables are labeled with the tags `D` and `H`, respectively, the value of objective function is labeled with the tag `obj_fn,` and the values of the volume constraint are labeled with the tags `nln_con1` and `nln_con2`, respectively. Note that one of the constraint function values is initially violated (`< 0`) because the initial design parameters were not feasible. However, the numerical optimizer has the capability to find a design that is both feasible and optimal for this example.

Between the optimization variables and the function values the content of the system call to the simulator is displayed as `"(container /var/tmp/aaaa0041c /var/tmp/baaa0041c)"`, with `container` being the name of the simulator and `/var/tmp/aaaa0041c` and `/var/tmp/baaa0041c` being the path and names belonging to the DAKOTA-to-simulator input and simulator-to-DAKOTA output files, respectively. Temporary files have been used in this case and these are deleted as soon as the simulator-to-DAKOTA output file is read. However, it is also possible to specify that the i/o files are to be saved under user supplied names with DAKOTA generated tag extensions, see File saving on page 82 and File tagging on page 82 for more information.

Just preceding the output of the objective and constraint function values is the line "`Active set vector = { 1 1 1 }`". The **active set vector** is not to be confused with the active constraint set that is sometimes defined for numerical optimization algorithms. For this case the active set vector is used for a DAKOTA-to-simulator request, and indicates the type of request that has been made to the simulator for the objective and constraint function evaluations. The first value of `1` on this DAKOTA output line indicates that the simulator is to evaluate the objective function. The remaining values of `1` indicate that the simulator is to evaluate the volume constraint functions. Had a value of `0` appeared in any of these positions it would have been interpreted by the simulator as a do-not-evaluate request for the respective objective or constraint function. The values contained in this active set vector correspond to the numbers in the first character position of the last three lines of the DAKOTA-to-simulator input file described in the section titled Constructing the interface on page 23.

Since finite difference gradient computations have been specified DAKOTA computes their values, in part by automatically making additional function evaluation requests to the simulator. Examples of the gradient-related function evaluations have been included in the sample output,

beginning with the line that reads "`>>>>> Dakota finite difference evaluation for x[1] + h:`". A sample of the resulting objective and constraint function values and their gradients is shown following function evaluation `5` beginning with the line "`>>>>> Total response returned to iterator:`". Here, another type of active set vector is displayed in the DAKOTA output file. The line "`Active set vector = { 3 3 3 }`" displays a DAKOTA-to-numerical-optimizer active set vector. It indicates the values that DAKOTA is supplying to the numerical optimizer associated with the objective function and constraints. The values of `3` are composite combinations used to indicate that the results of a function evaluation, `1`, and a DAKOTA gradient computation, `2`, are being supplied to the numerical optimizer, for each of the objective and constraint functions. The composite values are computed by simple addition (`1+2=3`). Some numerical optimizers also request Hessian information. For this case a code of `4` is used. Thus, if the numerical optimizer were being supplied with function value and Hessian information the active set value would be `1+3=4` or if function value, gradient and Hessian information were being supplied the active set value would be `1+2+3=7,` for the associated objective or constraint function.

The final lines of the DAKOTA output, beginning with the line "`<<<<< Single method iteration completed`", summarize the results of the optimization analysis. The best values of the optimization parameters, objective function, and constraint equations are output. Since the analysis is approximate the constraint functions are only satisfied to within some small tolerance of zero for this example. The DAKOTA results are followed by a summary of the NPSOL analysis. A more detailed summary of the NPSOL analysis is contained in either file `fort.9` or file `ftn09`, as specified in the output.

Figure 7 Example DAKOTA output

```
MPI initialized with 1 processors.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
no method_pointer: last specifications parsed will be used
methodName = npsol_sqp
gradientType = numerical
hessianType = none
Numerical gradients using 0.1% central differences
to be calculated by the dakota finite difference routine.
Optimality Tolerance        = 0.0001
NOTE: NPSOL's convergence tolerance is not a relative tolerance.
      See pp. 21-22 of NPSOL manual for description.
Derivative Level            = 3
Running MPI executable in serial mode.
Running Single Method Strategy...

-------------------------------------
Begin Dakota finite difference routine
-------------------------------------

>>>>> Initial map for non-finite-differenced portion of response:

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      4.5000000000e+00 H
                      4.5000000000e+00 D

(container /var/tmp/aaaa0041c /var/tmp/baaa0041c)
Removing /var/tmp/aaaa0041c and /var/tmp/baaa0041c
```

```
Active response data for function evaluation 1:
Active set vector = { 1 1 1 }
                     1.0776762359e+02 obj_fn
                     8.0444076396e+00 nln_con1
                    -8.0444076396e+00 nln_con2


>>>>> Dakota finite difference evaluation for x[1] + h:

------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                     4.5045000000e+00 H
                     4.5000000000e+00 D

(container /var/tmp/caaa0041c /var/tmp/daaa0041c)
Removing /var/tmp/caaa0041c and /var/tmp/daaa0041c

Active response data for function evaluation 2:
Active set vector = { 1 1 1 }
                     1.0783442171e+02 obj_fn
                     8.1159770472e+00 nln_con1
                    -8.1159770472e+00 nln_con2

>>>>> Dakota finite difference evaluation for x[1] - h:

------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                     4.4955000000e+00 H
                     4.5000000000e+00 D

(container /var/tmp/eaaa0041c /var/tmp/faaa0041c)
Removing /var/tmp/eaaa0041c and /var/tmp/faaa0041c

Active response data for function evaluation 3:
Active set vector = { 1 1 1 }
                     1.0770082548e+02 obj_fn
                     7.9728382320e+00 nln_con1
                    -7.9728382320e+00 nln_con2


>>>>> Dakota finite difference evaluation for x[2] + h:

------------------------------
Begin Function Evaluation    4
------------------------------
Parameters for function evaluation 4:
                     4.5000000000e+00 H
                     4.5045000000e+00 D

(container /var/tmp/gaaa0041c /var/tmp/haaa0041c)
Removing /var/tmp/gaaa0041c and /var/tmp/haaa0041c

Active response data for function evaluation 4:
Active set vector = { 1 1 1 }
                     1.0791640170e+02 obj_fn
                     8.1876180243e+00 nln_con1
                    -8.1876180243e+00 nln_con2


>>>>> Dakota finite difference evaluation for x[2] - h:

------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                     4.5000000000e+00 H
                     4.4955000000e+00 D
```

```
(container /var/tmp/iaaa0041c /var/tmp/jaaa0041c)
Removing /var/tmp/iaaa0041c and /var/tmp/jaaa0041c

Active response data for function evaluation 5:
Active set vector = { 1 1 1 }
                     1.0761892743e+02 obj_fn
                     7.9013403937e+00 nln_con1
                    -7.9013403937e+00 nln_con2


>>>>> Total response returned to iterator:

Active set vector = { 3 3 3 }
                     1.0776762359e+02 obj_fn
                     8.0444076396e+00 nln_con1
                    -8.0444076396e+00 nln_con2
 [  1.4844025288e+01  3.3052696308e+01  ] obj_fn gradient
 [  1.5904312809e+01  3.1808625618e+01  ] nln_con1 gradient
 [ -1.5904312809e+01 -3.1808625618e+01  ] nln_con2 gradient

.
.
.

>>>>> Dakota finite difference evaluation for x[2] - h:

------------------------------
Begin Function Evaluation   40
------------------------------
Parameters for function evaluation 40:
                     4.9556729812e+00 H
                     4.0359108491e+00 D

(container /var/tmp/adaa0041c /var/tmp/bdaa0041c)
Removing /var/tmp/adaa0041c and /var/tmp/bdaa0041c

Active response data for function evaluation 40:
Active set vector = { 1 1 1 }
                     9.8930418512e+01 obj_fn
                    -1.2698647482e-01 nln_con1
                     1.2698647482e-01 nln_con2


>>>>> Total response returned to iterator:

Active set vector = { 3 3 3 }
                     9.9062468783e+01 obj_fn
                     1.8074075570e-10 nln_con1
                    -1.8074075570e-10 nln_con2
 [  1.3326473792e+01  3.2694282247e+01  ] obj_fn gradient
 [  1.2818642490e+01  3.1448402789e+01  ] nln_con1 gradient
 [ -1.2818642490e+01 -3.1448402789e+01  ] nln_con2 gradient



NPSOL exits with INFORM code = 0 (see p. 8 of NPSOL manual)

NOTE: see Fortran device 9 file (fort.9 or ftn09)
      for complete NPSOL iteration history.

<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 40 total (40 new, 0 duplicate)
<<<<< Best design parameters  =
                     4.9556729812e+00 H
                     4.0399507999e+00 D
<<<<< Best objective function =
                     9.9062468783e+01
<<<<< Best constraint values  =
                     1.8074075570e-10
                    -1.8074075570e-10
Run time from MPI_Init to MPI_Finalize is 6.0880220000e+00 seconds
```

```
                    NPSOL  ---  Version 4.06-2     Nov  1992
                    ======================================

Maj  Mnr    Step  Fun  Merit function  Violtn Norm gZ   nZ Penalty Conv
   0    3 0.0E+00    1  1.07767624E+02 1.1E+01 1.5E+00    1 0.0E+00 F FF
   1    1 1.0E+00    2  9.95643509E+01 4.2E+00 1.3E+00    1 0.0E+00 F FF
   2    1 1.0E+00    3  9.91019314E+01 6.5E-01 3.8E-01    1 0.0E+00 F TF
   3    1 1.0E+00    4  9.90642035E+01 1.3E-01 9.4E-02    1 0.0E+00 F TF
   4    1 1.0E+00    5  9.90624728E+01 5.2E-03 3.6E-03    1 0.0E+00 T TF
   5    1 1.0E+00    6  9.90624688E+01 6.4E-06 1.8E-04    1 0.0E+00 T TF
   6    1 1.0E+00    7  9.90624688E+01 1.9E-08 4.1E-06    1 0.0E+00 T TF
   7    0 1.0E+00    8  9.90624688E+01 2.6E-10 5.2E-12    1 0.0E+00 T TT


 Exit NPSOL - Optimal solution found.

 Final nonlinear objective value =    99.06247
```

Some useful features of DAKOTA

DAKOTA has many features that can be used to enhance your problem solving capability, including ones that can be used to reduce the overall amount of time you could spend running an analysis. Some of these features are implicit to the DAKOTA input file, since this file allows you to readily change between analysis types, vendor codes, application problems, etc. Other useful time-saving features are also present in DAKOTA. In this section examples of the restart capability and the parallel processing interface will be discussed.

Restarting DAKOTA

DAKOTA was developed for solving problems that typically require multiple calls to computationally expensive simulation codes. In some cases you may want to conduct the same optimization, but to a finer final convergence tolerance. This would be costly if the entire optimization analysis had to be repeated. Power outages and system failures could also result in costly delays. However, DAKOTA automatically records enough of the input and response data from calls to your simulation code so that a time-inexpensive restart is possible.

As an example of the DAKOTA restart capability, consider the above container example again. For the sake of this example, pretend that the simulator function evaluations are expensive and that the DAKOTA run unexpectedly aborted after 20 successful iterations. Assuming that the original DAKOTA analysis was started with the command

```
dakota -i dakota_container.in
```

DAKOTA will automatically generate a file named `dakota.rst` that contains input and response information from the aborted run. To instruct DAKOTA to essentially "pick up where it left off" execute the command

```
dakota -i dakota_container.in -r -s 20 -w dakota_new.rst
```

This command tells DAKOTA to recover the results of the first twenty simulator calls from the restart file and then proceed with the analysis by making simulator calls as usual, writing the new

restart file `dakota_new.rst`. A more in depth discussion of the restart capability with additional features is given in Restart Management on page 125.

The parallel interface

If you have more than one processor available, such as a cluster of network-connected workstations or a multi-processor, then the solution time required for a DAKOTA analysis can often be substantially reduced through use of parallel distributed processing techniques. For many of the optimization and other methods supported by DAKOTA, parallel processing can dramatically reduce analysis times when the simulator function evaluations are computationally expensive. The reason behind this is that many of these methods contain at least some independent calls to the simulator which can be distributed between processors on every iteration step. If a given method has `n` independent simulator calls at every iteration step then the DAKOTA analysis speed can be increased by as much as a factor of `n` by running multiple instances of the simulator, one on each processor. For maximum speed increase, it has been assumed that at least `n` processors are available to DAKOTA for simulator evaluations and that the computation time for an individual simulator call is suitably high (typically on the order of a few seconds or less for modern workstation clusters) so that interprocessor communication time is a negligible in comparison. Performance increases can still be obtained for systems with fewer than `n` processors.

DAKOTA has been developed with parallel processing capabilities built into its framework. Thus, if you have a new or existing application that could benefit from making parallel simulator calls, DAKOTA allows you to exploit parallelism with the addition of only a few commands to the dakota input file and some minor changes to the command line. DAKOTA can also be used in conjunction with simulators that have their own parallel capabilities. For a complete description of the parallel capabilities associated with DAKOTA see Exploiting Parallelism on page 99.

This section will explain where parallelism is exploited in typical optimization algorithms and show how to set up and run a simple DAKOTA optimization analysis using parallel processing techniques. It is assumed that DAKOTA has been configured with the MPI package as described in Configuring with the Message Passing Interface on page 184. Here, the previously defined container optimization example will be extended to allow parallel processing of the finite difference gradient computations. For further examples of incorporating parallelism into a DAKOTA analysis see Specifying Parallelism on page 107.

Gradient based local optimization algorithms typically consist of an initialization phase followed by an iterative phase, where each iteration consists of: the computation of a search direction in the multi-dimensional parameter space, a search along the established direction for a sufficient decrease in the objective function (subject to any constraints that may be present), a gradient computation, an update to a matrix approximating the second partial derivatives (Hessian) of the constrained objective, and a convergence check. There are many opportunities to exploit parallelism in this type of algorithm. However, not all these opportunities would turn out to be productive in light of the fact that the simulator calls usually dominate the overall computational effort.

The search direction computation is based on the Hessian approximation and the gradient from the previous iteration or from the initialization phase. The objective and constraint function values, gradients, and the Hessian approximation are used to compute the search direction. This direction points to the minimum value of the current estimate of the optimization problem that satisfies the constraints. This subproblem is only an approximation to the actual nonlinear optimization problem, and thus, the overall optimization algorithm must proceed iterative manner to a solution. The search direction computation is based on linear algebra and the computational effort expended is usually very small in comparison to the simulator calls. This conclusion also holds true for other parts of the optimization algorithm algebra, such as the update to the Hessian approximation. The use of parallel processing to solve the optimization algebra is not typically advantageous unless the number of optimization parameters is huge and the simulator function evaluations are relatively inexpensive. The development of this type of parallelism is also strongly tied to the internal data structures of the optimizer. For these reasons, this form of parallelism is not directly supported by DAKOTA. However, it is possible to link an optimizer with these capabilities to DAKOTA should the need arise.

The part of the problem where it is advantageous to utilize parallel processing is where multiple calls to the simulator evaluator can be made in parallel. For gradient-based optimization, this opportunity occurs during the line search and gradient computation steps. During these steps both function and gradient information for the constraint and objective functions are computed. For some types of line search, the gradient is computed directly after the completion of the line search. For other cases it is an integral part of the line search. For either type of line search, the gradient information can be computed on additional processors at the same time as the objective and constraint function values are computed. For the container optimization example if central finite differences are used in the gradient computations, then an additional four gradient-related simulator evaluations can be performed on four additional processors. For expensive simulator evaluations, this would result in a maximum speed increase of a factor of five.

Enabling parallel optimization capabilities in DAKOTA is quite easy. The container optimization problem will be used as an example. While the container simulator function calls are quite inexpensive in actuality, it is used here for the sake of example. The general set up for a simulator with expensive function evaluations would follow along the same lines and the output obtained would be much the same.

No changes are necessary between the DAKOTA to interface input code for serial and parallel analyses. Some minor changes may be necessary for the interface to DAKOTA output code for the parallel analysis. The reason for this is that the current version of DAKOTA operating in parallel mode polls for the existence of the interface-to-DAKOTA output file and once its existence is detected a read attempt is made. However, it may be that the interface is not finished writing this file and therefore the read attempt will fail. This condition can occur, for example, when there is a large amount of output, when a computationally expensive interface alternates between calculation and output operations, or when there are write delays due to heavy system loading. DAKOTA has the capability to recover from up to ten failed read attempts of this type on any interface-to-DAKOTA input file, but the potential for this condition can often be avoided

entirely by making some simple changes to the simulator output procedures. The approach used here is to write the simulator to DAKOTA output to a uniquely named temporary file, and when all the output has been written and this file has been closed, move or rename it to the file name stored in `outfile`. Other possibilities exist, and are discussed in System call synchronization on page 101.

The temporary file name can be generated in a variety of ways. However, care must be taken so that each simulator that is in operation uses a different name. For the container example, that would require having five different file names on each iteration. One approach to generating unique file names would be to add one or more characters to the name stored in `outfile`. However, although such an occurrence would be unlikely, there is no guarantee that this would produce file names that are not already in use somewhere else. Another approach would be to obtain a unique file name using scratch files in Fortran or from the C-library function `tmpnam` in C or C++.

For the Fortran version the `open` statement the listing in Figure 3 is replaced by

```
open(11,STATUS='SCRATCH')
inquire(11,NAME=tmpfile)
```

where `tmpfile` is a character variable of the appropriate dimension. Write the output data to this file and replace the `close` statement inFigure 3 with

```
close(unit=11,STATUS='KEEP')
```

The `tmpfile` is moved to `outfile` with the statements

```
sysvar = "mv " // tmpfile // " " // outfile
call system(sysvar)$DAKOTA/test/container_p.f
```

The code for the parallel version is located in file `container_p.f` in the `$DAKOTA/test/` directory. This version is not compatible with silicon graphics platforms, which do not allow closing a scratch file with `STATUS='KEEP'`. For this case an alternative mixed language version that calls `tmpnam` is located in files `container_p2.f` and `tempnm.c.` To compile the Fortran version `container_p.f` you will need to enter something like

```
f77 -o container_p container_p.f
```

or for the mixed language version

```
cc -c tempnm.c
f77 -o container_p container_p2.f tempnm.o
```

For the C and C++ versions a temporary file name is obtained with the function call

```
tmpnam(tmpfile);
```

The file `tmpfile` is opened, response data is written, and it is closed according to standard C or C++ conventions. The file is then moved to `outfile` using a system function call. The C and C++ versions are stored in files `container_p.c` and `container_p.C` in the `$DAKOTA/`

`test/` directory, respectively. To compile use commands similar to those given for the serial C and C++ versions.

These files are the same as the serial versions, with exception to the changes discussed. In the event that the simulator code is not directly accessible, the 3-piece Interface on page 81 can be used to incorporate the above file renaming strategy. It should be noted for problems that execute as fast as the container example, it is unlikely that a failure due to the race condition would occur in actuality. However, in any problem where significant delays can occur between the creation of the interface-to-DAKOTA response file and its completion, such a strategy is necessary. Also, if the simulator is compiled for use in a multi-thread environment then the `system` call in the Fortran version and the C call to `tmpnam` may not be suitable on some platforms unless re-entrant versions are available. For this case some other method should be used to avoid the race condition or the 3-piece interface could be used. For other approaches to avoiding the race condition see the discussion in System call synchronization on page 101.

If the parallel container optimization example is to be run on a cluster of network-connected workstations in master-slave mode under MPI, then only a few changes are necessary to the DAKOTA input file `dakota_container.in` in Figure 6. The name of the analysis driver in the interface specification must be set to `container_p`, the name of the parallel simulator executable. The command `parallel_library mpi` must be set in the strategy specification to request MPI as the parallel communication handler, and `evaluations asynchronous` must be set in the method specification to enable distributed parallel computation of the simulator function evaluations. These changes are shown in Figure 8 and are stored in file `dakota_container_p.in`.

Figure 8 DAKOTA input file for the parallel container optimization example.

```
Interface specification
        interface,                                          \
          application, system                               \
          analysis_driver = 'container_p'

# Variables specification
        variables,                                          \
          continuous_design = 2                             \
          cdv_descriptor 'H' 'D'                            \
          cdv_initial_point 4.5 4.5                         \
          cdv_lower_bounds 0.0 0.0

# Strategy specification
        strategy,                                           \
          single_method                                     \
          parallel_library mpi

# Method specification
        method,                                             \
          npsol_sqp                                         \
          evaluations asynchronous

# Responses speification
        responses,                                          \
          num_objective_functions = 1                       \
          num_nonlinear_constraints = 2                     \
          numerical_gradients                               \
          method_source dakota                              \
          interval_type central                             \
```

```
fd_step_size = 0.001                                    \
no_hessians
```

Another possibility for the avoidance of the file read race condition makes use of DAKOTA file File tagging on page 82 and UNIX shell scripting. For this approach, tagged file names are used in to eliminate write conflicts when multiple instances of the interface are running in parallel, and in the naming of temporary working directories. Shell scripts are used to actually create temporary working directories for individual instances of the interface, which eliminates the read race condition. File tagging is enabled by adding the commands

```
parameters_file=        'container.in'\
results_file=           'container.out'\
file_tag\
```

under the `interface` specification in Figure 8, and the analysis driver specification becomes

```
analysis_driver = 'container.script'\
```

One of the serial container executables listed in Figure 3 through Figure 5 is used with the shell scripting approach. The shell script file listing is given in Figure 9.

Figure 9 UNIX shell script file for parallel DAKOTA.

```
#! /bin/csh -f
# $argv[1] is container.in.(fn_eval_num) FROM Dakota
# $argv[2] is container.out.(fn_eval_num) returned to Dakota

# create a unique temporary directory using $argv[1]
set num = 'echo $argv[1] | cut -c 14-'
mkdir workdir.$num

#make workdir.$argv[1] the current working directory
cp $argv[1] workdir.$num
cd workdir.$num

#run the container optimization interface from workdir.$argv[1]
../container $argv[1] $argv[2]

#move the completed output file to the dakota working directory
mv $argv[2] ../.

#remove the temporary working directory
cd ..
rm -rf workdir.$num
```

The shell script is store in file `container.script` the DAKOTA input file for stored in file `dakota_container_pss.in` in the `$DAKOTA/test` directory. Other parallel interface possibilities exist within DAKOTA, see Implementation of Parallelism on page 104.

To execute DAKOTA in parallel mode it must be run within the proper environment. To run on a workstation cluster under MPI, for example, you might enter the command

```
mpirun -np 5 dakota -i dakota_container.in > dakota_out
```

The exact command would depend on how MPI is installed on your system. For a more detailed discussion see Running a parallel DAKOTA job on page 110. The output results for the Fortran version of `container_p` are shown in Figure 10. The parallel results are much the same as the serial results. The output file contains several lines indicating that DAKOTA is being run in a master-slave parallel mode and that the simulator function evaluations are being distributed over the slave servers. For this example a total of six processors are used. One processor acting as the master runs DAKOTA, and the remaining processors act as slave servers by conducting simulator

evaluations when a request is made. If the number of processors is limited it is also possible instruct MPI to use one of the processors as both a master and slave. Since DAKOTA is not in itself computationally expensive the processor can be shared between DAKOTA and the simulator function evaluation without much performance loss.

Figure 10 Sample output results for a parallel DAKOTA run

```
MPI initialized with 6 processors.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
no method_pointer: last specifications parsed will be used
methodName = npsol_sqp
gradientType = numerical
hessianType = none
Numerical gradients using 0.1% central differences
to be calculated by the dakota finite difference routine.
Optimality Tolerance        = 0.0001
NOTE: NPSOL's convergence tolerance is not a relative tolerance.
      See pp. 21-22 of NPSOL manual for description.
Derivative Level            = 3
Running MPI executable in parallel master-slave mode.
numSlaveServers = 5 procsPerAnalysis = 1 procRemainder = 0 parallelismLevel = 1
Running Single Method Strategy...

-------------------------------------
Begin Dakota finite difference routine
-------------------------------------

>>>>> Initial map for non-finite-differenced portion of response:

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                     4.5000000000e+00 H
                     4.5000000000e+00 D

(Parallel job 1 added to message passing queue)


>>>>> Dakota finite difference evaluation for x[1] + h:

------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                     4.5045000000e+00 H
                     4.5000000000e+00 D

(Parallel job 2 added to message passing queue)

.
.
.

>>>>> Dakota finite difference evaluation for x[2] - h:

------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                     4.5000000000e+00 H
                     4.4955000000e+00 D

(Parallel job 5 added to message passing queue)



Synchronizing 5 asynchronous evaluations.
First pass: num_sends = 5
```

```
Master assigning fn. evaluation 1 to server 1
Master assigning fn. evaluation 2 to server 2
Master assigning fn. evaluation 3 to server 3
Master assigning fn. evaluation 4 to server 4
Master assigning fn. evaluation 5 to server 5
Waiting on all jobs.

Active response data for function evaluation 1:
Active set vector = { 1 1 1 }
                     1.0776762359e+02 obj_fn
                     8.0444076396e+00 nln_con1
                    -8.0444076396e+00 nln_con2


Active response data for function evaluation 2:
Active set vector = { 1 1 1 }
                     1.0783442171e+02 obj_fn
                     8.1159770472e+00 nln_con1
                    -8.1159770472e+00 nln_con2

.
.
.
Begin Function Evaluation   40
-----------------------------
Parameters for function evaluation 40:
                      4.9556729812e+00 H
                      4.0359108491e+00 D

(Parallel job 40 added to message passing queue)



Synchronizing 5 asynchronous evaluations.
First pass: num_sends = 5
Master assigning fn. evaluation 36 to server 1
Master assigning fn. evaluation 37 to server 2
Master assigning fn. evaluation 38 to server 3
Master assigning fn. evaluation 39 to server 4
Master assigning fn. evaluation 40 to server 5
Waiting on all jobs.

Active response data for function evaluation 36:
Active set vector = { 1 1 1 }
                     9.9062468783e+01 obj_fn
                     1.8074075570e-10 nln_con1
                    -1.8074075570e-10 nln_con2
.
.
.
```

Gradient-based optimization is only one type of DAKOTA analysis that lends well to parallelism. Many of the other methods supported by DAKOTA also can be run in a parallel environment due to the independence of multiple function evaluations inherent in their design. The Monte Carlo, coordinated pattern search, and genetic algorithms of SGOPT are further examples where substantial speed increases can be obtained in a parallel environment for computationally expensive simulator evaluations, due to the existence of independent function evaluation calls in each algorithm. A complete list of DAKOTA methods for which parallel analysis can be used is given in Specifying Parallelism on page 107.

Decision Tables for DAKOTA Methods and Strategies

DAKOTA provides easy access to a large number of methods and strategies of varying capabilities. These individual methods and strategies can be looked at as modular components, any one of which may be applied in an overall analysis. As a combined resource, these modules can be used to solve a wide range of individual problem types. Knowing when and where to use particular methods and/or strategies will enhance the power and performance of DAKOTA, and give you a greater level of insight into your analysis. This section will be primarily concerned with the classification of optimization methods and strategies that are part of DAKOTA since they are many and varied. Nondeterministic methods and parameter studies will also be discussed.

Optimization algorithms can be categorized by several different means of classification, according to the uses for which they were designed. Whether the optimizer is for continuous, discrete, or mixed parameters; is unconstrained or constrained; has a single optimal solution or multiple possibilities; or has smooth or nonsmooth objective and constraint functions are some examples. As a first pass, several general types of classifications will be given and the associated methods will be categorized in tabular form.

The types of constraints that an optimization algorithm is designed to handle is one means of classification. Optimization algortihms are typically designed for use on problems without constraints (unconstrained optimizers), or designed so that they can handle upper and lower bound constraints on the optimization parameters, linear constraint functions, or nonlinear constraint functions. Categorization of the DAKOTA methods under the constraint classification is given in Table 1.

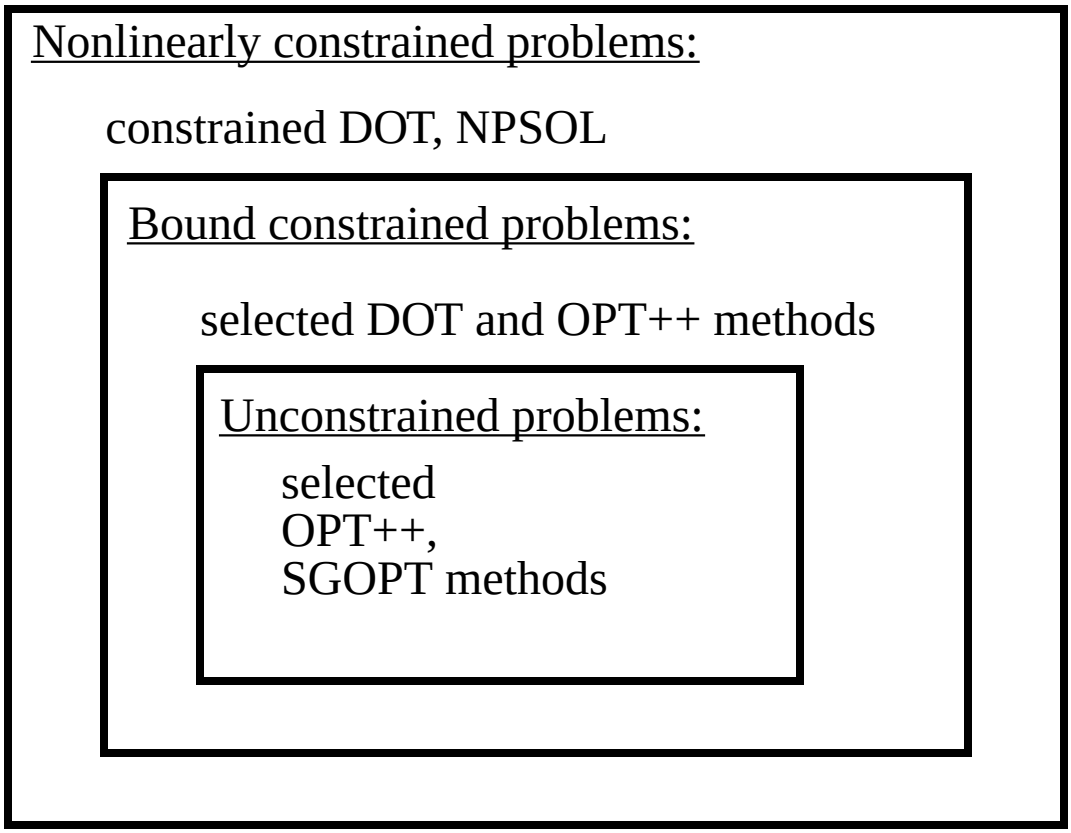
Table 1 Constraints

Constraints	Applicable Methods
unconstrained	optpp_cg, optpp_fd_newton, optpp_g_newton, optpp_newton, optpp_q_newton, most sgopt methods
bound constrained	dot_bfgs, dot_frcg, optpp_baq_newton, optpp_bc_elipsoid, optpp_bc_newton, optpp_bcg_newton, optpp_bcq_newton, sgopt_pga_real, sgopt_coord_ps
linearly constrained	special handling with npsol_sqp; otherwise any nonlinearly constrained method
nonlinearly constrained	npsol_sqp, dot_mmfd, dot_slp, dot_sqp

Constrained optimization algorithms are often designed as generalizations of unconstrained methods. This concept also holds between the different types of constrained optimizers, i.e. nonlinearly constrained is often a generalization of linearly constrained, which is a generalization of bound constrained. Thus, little or no performance loss would be observed for similar methods when a constrained version is applied to an unconstrained problem, etc. This concept is reflected

in Figure 11 where each generalization of the constraint type encompasses previous constraint types. This type of performance is particularly true of the gradient-based optimizers.

Figure 11 Generalizations of optimizer constraint handling capabilities.



The type of variable that an optimization code can operate on is another method of classification. Optimization codes designed to handle continuous or real-valued variables are the most prevalent in DAKOTA. Optimization codes that accept integer or a mix of real and integer variables are also accessible from DAKOTA, as well as codes that accept continuous nondeterministic variables. Table 2 categorizes the DAKOTA methods under the variables classification.

Table 2 Variables

Variables	Applicable Methods
continuous	DOT, NPSOL, and OPT++ methods, sao, sgopt_coord_ps, sgopt_coord_sps, sgopt_solis_wets, sgopt_pga_real, sgopt_strat_mc
discrete	sgopt_pga_int
mixed	sgopt_pga_mixed, branch_and_bound

Optimization problems involving minimization of strictly convex (i.e. bowl shaped) objective functions that are either unconstrained or have linear constraints have at most a single local optimal solution. However, minimization problems involving nonlinear constraints and/or nonconvex objective functions may have multiple local optimal solutions. Similar conclusions can be drawn for maximization problems. Algorithms that are designed to solve local optimization problems are typically much more efficient in terms of analysis time than ones that apply to global optimization problems, because they usually require vastly fewer function evaluations. However, it is often unknown whether the problem is global or local a priori. Thus, it is often necessary to apply a less efficient global optimization algorithm. The available DAKOTA methods are categorized as global or local in Table 3.

A procedure for determining whether a problem is best suited for global or local optimization can be somewhat of an art form. If the objective and constraint functions are not known analytically, then it is unlikely that it will be possible to make a judgement without further information. In some cases it may be desirable to combine global and local optimizers in a hybrid strategy in order to exploit the respective advantages of each, or to make some preliminary assessment of the objective and constraint function behaviors over the parameter space. DAKOTA provides methods and strategies for performing these types of analyses. See Multilevel Hybrid Optimization on page 71 and Parameter Study Capabilities on page 62 for more details.

Table 3 Local vs. global

Solution Type	Applicable Methods
local	DOT, NPSOL, and OPT++ methods (except optpp_pds), sao, sgopt_coord_ps, sgopt_coord_sps, sgopt_solis_wets
global	optpp_pds, sgopt_pga_real, sgopt_pga_int, sgopt_strat_mc

Optimization algorithms that have been designed to operate on smooth functions can sometimes suffer severe performance losses if the problems that they are applied to are actually nonsmooth. Table 4 categorizes DAKOTA methods as being suitable for either smooth or nonsmooth analysis. The term smooth is often used to describe functions that have theoretically continuous gradient and Hessian information. It can be noted that by this definition, numerical analysis is nonsmooth whenever finite precision arithmetic is used. However, in practice all the methods employed by DAKOTA can tolerate at least some degree of nonsmoothness. What differentiates between the categories of smooth and nonsmooth here is whether or not they are immune to relatively high levels of nonsmoothness.

Gradient based methods cannot tolerate high levels of nonsmoothness, and thus they comprise the smooth optimization category. Limiting their use to relatively smooth functions is especially important when finite differencing is used to compute the gradients. However, if the nonsmoothness is small in comparison to changes that can be observed in the objective function over some parameter range, then they may be suitable for use. For this case methods intended for smooth optimization could provide a relatively fast means of obtaining large improvements in the objective function value. However, convergence to an optimal point can not be guaranteed, and if finite differencing were employed a relatively large step size would be needed.

Determining when a smooth method is acceptable for use on a given optimization problem, is again, somewhat of an art form. It may be necessary to gain insight into the level of nonsmoothness present through use of DAKOTA's Parameter Study Capabilities on page 62. As a rule of thumb, the finite difference step size should be set so that level nonsmoothness in the neighborhood initial point is no more than ~10% of the net change in the objective function in the same neighborhood. It should also be apparent that the net observed change in the objective function is a large scale change, rather than some form of local waviness. Similar considerations should be made for the constraint functions. A close observation of the optimization results usually reveals that much more work is being performed in the line search part of the optimization algorithm for nonsmooth problems. However, the total work performed is usually much less for than would be observed for a nonsmooth optimization code. This analysis could be followed up by one or more of the nonsmooth optimization methods if further improvement in the objective function is needed.

Table 4 Smooth vs. nonsmooth

Function Surface	Applicable Methods
smooth	gradient-based: DOT, NPSOL, OPT++ methods (except optpp_pds)
nonsmooth	optpp_pds, sao, sgopt_coord_ps, sgopt_coord_sps, sgopt_solis_wets, sgopt_pga_int, sgopt_pga_real, sgopt_strat_mc

If you have access to a cluster of network-connected workstations or a multiprocessor machine, then you can exploit parallelism in the execution of your optimization problem to reduce the overall analysis time. Given that the function evaluations are expensive, algorithmic coarse-grained parallelism can be exploited in cases where multiple independent function evaluations are made by the optimization code. All the methods supported by DAKOTA support at least some algorithmic coarse grained parallelism in one or more specific operating modes. Table 5 categorizes the algorithms. The gradient-based optimizers support speculative analysis in some modes. For this method DAKOTA speculates that gradient information will be requested by the optimization algorithm soon after a function evaluation request is made. By computing gradient information in parallel, at the same time as the function evaluation, a reduction in the overall analysis time is achieved. However, the gradient information may not be used by the optimization program on every iteration. A more general form of parallelism is supported by some of the gradient-based and all the other types of optimization programs. For these codes, multiple independent function evaluations are always requested on every iteration, and thus the speculative nature is not present.

Table 5 Algorithmic parallelism

Parallelism	Applicable Methods
Serial	standard DOT, NPSOL, and OPT++ methods using analytic and vendor numerical gradients
Parallel	DOT, NPSOL, and OPT++ methods using DAKOTA numerical gradients, optpp_pds, SGOPT methods

Other classifications are also important. For instance, when function evaluations become extremely expensive, methods that typically require tens of thousands of function evaluations such as genetic algorithms or Monte Carlo analysis must be ruled out unless a large parallel machine is available. The number of optimization parameters can also be a factor. For nongradient-based methods, the probability of finding an improved objective function value on the next iteration step falls off quickly as the problem dimension increases. This is true even if the number of processors is scaled with the problem dimension.

Table 6 summarizes the previous classifications. Blank entries in a given column inherit the category from the previous row.

Table 6 All inclusive summary

Variable Type	Function Surface	Solution Type	Constraints	Applicable Methods
continuous	smooth	local	unconstrained	optpp_cg, optpp_fd_newton, optpp_g_newton, optpp_newton, optpp_q_newton
			bound constrained	dot_bfgs, dot_frcg, optpp_baq_newton, optpp_bc_elipsoid, optpp_bc_newton, optpp_bcg_newton, optpp_bcq_newton
			nonlinearly constrained	npsol_sqp, dot_mmfd, dot_slp, dot_sqp
	nonsmooth	local	bound constrained	sgopt_coord_ps, sgopt_coord_sps, sgopt_solis_wets
			dependent on underlying optimizer	sao
		global	bound constrained	sgopt_pga_real, sgopt_strat_mc
			nonlinearly constrained	(coming soon: sgopt_pga_real)
discrete	n/a	global	bound constrained	sgopt_pga_int
mixed	smooth	local	nonlinearly constrained	branch_and_bound
	nonsmooth	global	bound constrained	sgopt_pga_mixed, (coming soon: sgopt_pga_mixed)

DAKOTA supports interfacing with a number of methods that are not directly used for optimization, and several strategies that incorporate optimization methods. Some of these have already been mentioned. These additional capabilities are divided into nondeterministic analysis, parameter study, and optimization strategy categories in Table 7.

Table 7 Other method and strategy classifications

General classification	Applicable Methods
nondeterministic	nond_probability, nond_mean_value
parameter study	centered_parameter_study, list_parameter_study, multidim_parameter_study, vector_parameter_study
strategies	branch_and_bound, multi_level, ouu, sao

Iterator and Strategy Hierarchies

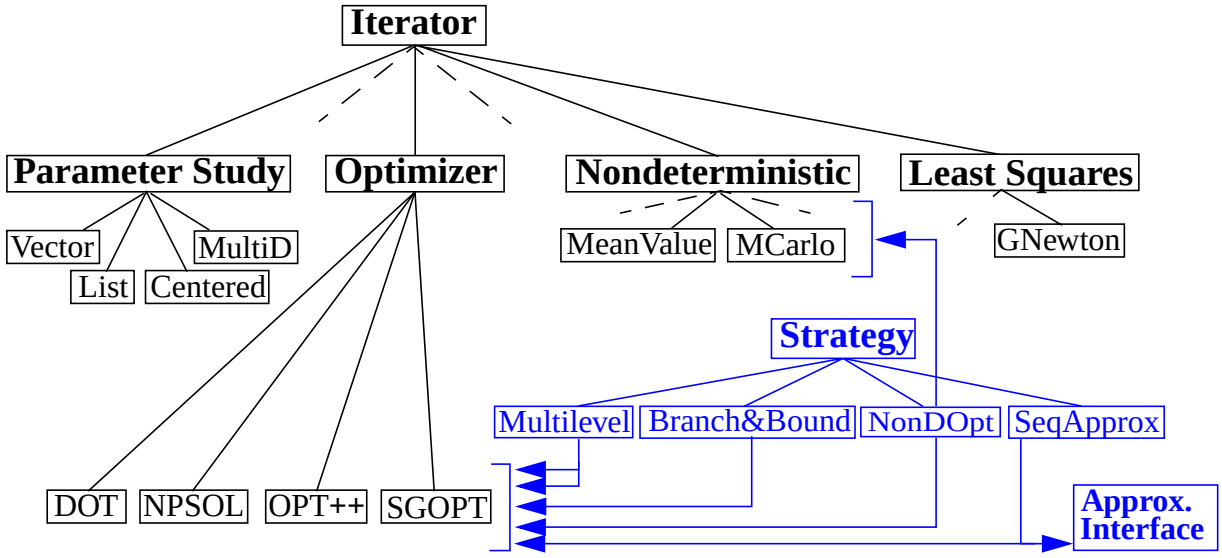


Figure 12 Iterator and Strategy Hierarchies

The DAKOTA system is designed to accommodate optimization, nondeterministic simulation, nonlinear least squares, and parameter study methods in its “iterator” hierarchy. These capabilities often complement each other in a project: (1) a parameter study is used to investigate local design space issues in order to help select the appropriate optimizer and optimizer controls, (2) optimization is used to find a best design, and (3) nondeterministic simulation is used to assess the affects of parameter uncertainty on the performance of the optimal design. Other classes of iterator methods may be added as they are envisioned, which “leverages” the utility of the interface developments. For example, software effort in coordinating multiple instances of parallel simulations on a massively parallel computer (see Multilevel parallelism on page 103) is reusable among all of the iterators in the DAKOTA system. The inheritance hierarchy of these iterators is shown in Figure 12. Inheritance enables direct hierarchical classification of iterators and exploits their commonality by limiting the individual coding which must be done to only those features which make each iterator unique.

The iterator hierarchy is currently divided into four branches: the optimizer branch contains optimization algorithms from the DOT, NPSOL, OPT++, and SGOPT libraries, the nondeterministic branch implements Mean Value and Monte Carlo sampling (MCarlo) methods, the least squares branch incorporates a Gauss-Newton method (GNewton) from the OPT++ library, and the parameter study branch implements vector, list, centered, and multidimensional parameter study methods. Refer to the overviews describing Optimization Capabilities on page 54, Uncertainty Assessment Capabilities on page 58, Nonlinear Least Squares Capabilities on

page 60, and Parameter Study Capabilities on page 62 for more information on these iterator branches, and refer to Method Commands on page 156 for information on iterator specification.

The strategy class hierarchy implements a variety of advanced approaches in which multiple iterators from the iterator hierarchy can be instantiated and bound to multiple models. These strategies coordinate multiple levels of iteration, monitor performance, and adapt iterators and models (switch/refine control) based on observed performance. In addition, strategies manage the distribution of tasks between the master and slave processors in implementing parallelism (see Exploiting Parallelism on page 99). The multilevel hybrid strategy uses multiple optimizers in succession with the best point from one iterator being used as the starting point for a subsequent iterator. The single method strategy (not shown) invokes a single iterator using a single model and can be viewed as a strategy layer bypass. The branch and bound strategy is under development for solution of mixed continuous/discrete applications. The optimization under uncertainty strategy incorporates an uncertainty quantification within the optimization process. And, in the sequential approximate optimization strategy, an optimizer is interfaced with an approximate design space representation from the hierarchy described in The Approximation Interface on page 95. Refer to the overview of Strategy Capabilities on page 70 for more information on strategy concepts and procedures, and refer to Strategy Commands on page 150 for information on strategy specification.

Optimization Capabilities

Introduction

Optimization methods in the DAKOTA system involve the manipulation of objective and constraint functions and potentially their gradient vectors and Hessian matrices. Currently, the number of objective functions must be `1`, since multi-objective optimization formulations are not yet explicitly supported. Thus the `m` functions in the DAKOTA response data set are interpreted as `1` objective function and `m-1` constraint functions within the DAKOTA optimizer hierarchy.

Some optimizers (e.g., NPSOL) have the ability to distinguish constraints which are linear with respect to the design variables from those which are nonlinear. In the linear case, a single matrix containing the coefficients of the linear constraint terms is sufficient to define the values of these constraints for all parameter sets. By providing this matrix to an optimizer which supports special handling of linear constraints, it becomes unnecessary for the user to evaluate these constraints on every function evaluation since the optimizer will evaluate them internally (see [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986]). However, since most engineering applications involve nonlinear contraints which are implicit functions of the design variables, a mechanism for specification of this linear constraint matrix has not yet been developed within DAKOTA. That is, special handling of linear constraints is not yet supported and linear constraints should be treated as general nonlinear constraints (evaluated on every function evaluation).

In DAKOTA, all nonlinear constraints are inequality constraints of the form $g_i(X) \leq 0$. Therefore, constraints of the form $c(X) \geq 0$ must be converted to the form $-c(X) \leq 0$. Furthermore, each equality constraint $h(X) = 0$ must be implemented by two oppositely signed inequality constraints: $h(X) \leq 0$ and $-h(X) \leq 0$.

When gradient and/or Hessian information is used in the optimization, it is assumed that derivative components will be computed only with respect to the *continuous design variables*. The omission of discrete variables from gradient vectors and Hessian matrices is common among all iterators (since derivatives with respect to discrete variables do not exist); however, inclusion of only the continuous design variables differs from parameter study iterators (which assume derivatives with respect to all continuous variables) and from nondeterministic analysis iterators (which assume derivatives with respect to the uncertain variables).

DOT Library

The DOT library [Vanderplaats Research and Development, 1995] contains nonlinear programming optimizers, specifically the Broyden-Fletcher-Goldfarb-Shanno (DAKOTA's `dot_bfgs` method) and Fletcher-Reeves conjugate gradient (DAKOTA's `dot_frcg` method) methods for unconstrained optimization, and the modified method of feasible directions

(DAKOTA's `dot_mmfd` method), sequential linear programming (DAKOTA's `dot_slp` method), and sequential quadratic programming (DAKOTA's `dot_sqp` method) methods for constrained optimization.

All DOT methods are local gradient-based optimizers which are best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the NPSOL Library on page 55.

DAKOTA controls the maximum number of iterations and function evaluations, the convergence tolerance, the output verbosity, and the optimization type for the DOT methods from its input specification. See DOT Methods on page 161 for additional details on DOT method specifications.

NPSOL Library

The NPSOL library [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] contains a sequential quadratic programming (SQP) implementation (DAKOTA's `npsol_sqp` method). SQP is a nonlinear programming optimizer for constrained minimization.

NPSOL's local gradient-based optimizer is best suited for efficient navigation to a local minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. Other gradient based optimizers for constrained optimization include the DOT Library on page 54.

DAKOTA controls the maximum number of iterations and function evaluations, the convergence tolerance, the output verbosity, the verification level, the function precision, and the line search tolerance for NPSOL from its input specification. See NPSOL Method on page 162 for additional details on NPSOL specifications.

The NPSOL library generates diagnostics in addition to those appearing in the DAKOTA output stream. These diagnostics are written to the default FORTRAN device 9 file (e.g., `fort.9` on the Sun Solaris architecture) in the working directory.

OPT++ Library

The OPT++ library [Meza, J.C., 1994] contains primarily nonlinear programming optimizers for unconstrained minimization: Polak-Ribiere conjugate gradient (DAKOTA's `optpp_cg` method), quasi-Newton, barrier function quasi-Newton, and bound constrained quasi-Newton (DAKOTA's `optpp_q_newton`, `optpp_baq_newton`, and `optpp_bcq_newton` methods), Gauss-Newton and bound constrained Gauss-Newton (DAKOTA's `optpp_g_newton` and `optpp_bcg_newton` methods - part of DAKOTA's nonlinear least squares branch), full Newton, barrier function full Newton, and bound constrained full Newton (DAKOTA's

optpp_newton, optpp_ba_newton, and optpp_bc_newton methods), finite difference Newton (DAKOTA's optpp_fd_newton method), and bound constrained ellipsoid (DAKOTA's optpp_bc_ellipsoid method). The library also contains the PDS nongradient-based method (parallel direct search [Dennis, J.E., and Torczon, V.J., 1994], specified as DAKOTA's optpp_pds method), and an input place holder for new algorithm testing (DAKOTA's optpp_test_new method).

OPT++'s gradient-based optimizers are best suited for efficient navigation to a local unconstrained minimum in the vicinity of the initial point. Global optima in nonconvex design spaces may be missed. OPT++'s PDS method does not use gradients and has some limited global identification abilities; it is best suited for problems for which gradient information is unavailable or is of questionable accuracy due to numerical noise. Some OPT++ methods support bound constraints, but none currently support general linear and nonlinear constraints. For gradient-based optimization with constraints, the DOT Library on page 54 and the NPSOL Library on page 55 should be used. For OPT++'s least squares methods, refer to Gauss-Newton on page 61.

DAKOTA manages the following inputs for OPT++ methods from its input specification: the maximum number of iterations and function evaluations, the convergence tolerance, the output verbosity, the search method, the maximum step, the gradient tolerance, the initial radius for ellipsoid methods, and the search scheme size for PDS. See OPT++ Methods on page 165 for additional details on these specifications.

The OPT++ library generates diagnostics in addition to those appearing in the DAKOTA output stream. These diagnostics are written to the file OPT_DEFAULT.out in the working directory.

SGOPT Library

The SGOPT (Stochastic Global OPTimization) library [Hart, W.E., 1997] contains a variety of global optimization algorithms, with an emphasis on stochastic methods. SGOPT currently includes the following global optimization methods: real-valued and integer-valued genetic algorithms (sgopt_pga_real, sgopt_pga_int) and stratified Monte Carlo (sgopt_strat_mc). Evolutionary pattern search algorithms, simulated annealing, tabu search, and multistart local search (see The Coupled Approach on page 73) are global methods which are under development but are not available in DAKOTA V1.0. Additionally, SGOPT includes several local search algorithms such as Solis-Wets (sgopt_solis_wets) and coordinate pattern search (sgopt_coord_ps, sgopt_coord_sps).

For expensive optimization problems, SGOPT's global optimizers are best suited for identifying promising regions in the global design space. In multimodal design spaces, the combination of global identification (from SGOPT) with efficient local convergence (from DOT, NPSOL, or OPT++) can be highly effective. None of the SGOPT methods are gradient-based, which makes them appropriate for discrete and mixed variable problems as well as problems for which gradient information is unavailable or is of questionable accuracy due to numerical noise. No SGOPT methods currently support general linear and nonlinear constraints directly, although

penalty function formulations for nonlinear constraints have been employed with success [Ponslet, E.R., and Eldred, M.S., 1996].

DAKOTA manages the following inputs from its input specification for all of SGOPT's methods: maximum number of iterations, maximum number of function evaluations, convergence tolerance, output verbosity, evaluation synchronization, maximum number of CPU seconds, and solution accuracy. In addition, each method has a variety of settings which are specific to it alone. Refer to SGOPT Methods on page 168 for additional details on all of the SGOPT specifications.

Uncertainty Assessment Capabilities

Introduction

Uncertainty assessment methods (also referred to as nondeterministic analysis methods) in the DAKOTA system involve the computation of probability distributions for response functions based on sets of simulations taken from the specified probability distributions for uncertain parameters. Thus the `m` functions in the DAKOTA response data set are interpreted as `m` general response functions (with no distinction between functions as with objective and constraint functions in the optimizer branch) within the DAKOTA uncertainty assessment hierarchy.

Within the variables specification, uncertain variable descriptions are employed to define the parameter probability distributions (see Uncertain Variables on page 138).

When gradient and/or Hessian information is used in the uncertainty assessment, it is assumed that derivative components will be computed only with respect to the *uncertain variables* (where all uncertain variables are continuous). The omission of discrete variables from gradient vectors and Hessian matrices is common among all iterators (since derivatives with respect to discrete variables do not exist); however, inclusion of only the uncertain variables differs from parameter study iterators (which assume derivatives with respect to all continuous variables) and from optimization and least squares iterators (which assume derivatives with respect to the continuous design variables).

Monte Carlo Probability

The Monte Carlo probability iterator is selected using the `nond_probability` specification. This iterator performs sampling for different parameter observations within a specified parameter distribution in order to assess the distributions for response functions. Probability of occurrence is then assessed by comparing the response results against response thresholds.

All Monte Carlo methods are sampling methods which can be extremely expensive in terms of the number of required function evaluations need to generate converged statistics. A different nondeterministic approach that can be less computationally demanding is the mean value method (see Mean Value on page 59).

DAKOTA controls the observations, the random seed, the sample type (pure random or Latin Hypercube), and the response thresholds for the Monte Carlo Probability method from its input specification. See Monte Carlo Probability Method on page 175 for additional details on this method specification.

Mean Value

The mean value method is selected using the `nond_mean_value` specification. This iterator computes approximate response function distribution statistics based on specified parameter distributions. The mean value method is a direct analytical method and does not perform any random sampling.

Since the mean value method does not perform random sampling, it can be much less computationally demanding than the Monte Carlo approach (see Monte Carlo Probability on page 58). However, since the method is based on Gaussian distribution assumptions and linearizations, the accuracy of the statistics must be carefully evaluated.

DAKOTA controls the response file names for the mean value method from its input specification. See Mean Value Method on page 176 for additional details on this method specification.

Nonlinear Least Squares Capabilities

Gauss-Newton on page 61

Introduction

Nonlinear least squares methods in the DAKOTA system are optimizers which exploit the special structure of a least squares objective function. These problems commonly arise in parameter estimation and test/analysis reconciliation. In order to exploit the problem structure, response data at a “finer grain” are required. Rather than using the least squares objective function and its gradient, least squares iterators require each term in the sum-of-squares formulation along with its gradient as the data set returned by the simulation. This means that the `m` functions in the DAKOTA response data set consist of the individual terms in the sum-of-the-squares objective function, rather than an objective function and `m-1` constraint functions (as they are in the optimizer branch). These individual terms are often called residuals in cases where they denote errors of observed quantities from desired quantities. Refer to Rosenbrock Problem Formulation on page 204 for an example showing the relationship between optimization and least squares response functions.

This enhanced granularity allows for simplified computation of an approximate Hessian matrix which only uses residual derivative information, since terms in the Hessian matrix which contain residual second derivatives also contain the residuals themselves and will become negligible as the residuals tend towards zero. That is, residual function and gradient information is sufficient to define the value, gradient, and approximate Hessian of the least squares objective function.

In practice, least squares solvers will tend to be significantly more efficient than general-purpose optimization algorithms when the residuals tend towards zero at the solution. Least squares solvers may experience difficulty when the residuals at the solution are significant.

As for optimization iterators, it is assumed that gradient and/or Hessian information will be computed only with respect to the *continuous design variables*. The omission of discrete variables from gradient vectors and Hessian matrices is common among all iterators (since derivatives with respect to discrete variables do not exist); however, inclusion of only the continuous design variables differs from parameter study iterators (which assume derivatives with respect to all continuous variables) and from nondeterministic analysis iterators (which assume derivatives with respect to the uncertain variables).

In order to specify a least-squares problem, the responses section of the Dakota input should be configured using `num_least_squares_terms` to define the number of functions, using either `numerical_gradients`, `analytic_gradients`, or `mixed_gradients` to define the gradients of these least squares terms, and using `no_hessians`, since no Hessian will be supplied from the simulator (it will be approximated internally).

Gauss-Newton

Gauss-Newton iterators (DAKOTA's `optpp_g_newton` and `optpp_bcg_newton` methods) approximate the true Hessian matrix by neglecting terms in which the residual function values appear, under the assumption that the residuals tend towards zero at the solution. The Gauss-Newton algorithm is part of the OPT++ package [Meza, J.C., 1994]. For a more complete description of the OPT++ package, refer to OPT++ Library on page 55.

Gauss-Newton is a gradient-based algorithm and is best suited for efficient navigation to a local least squares solution in the vicinity of the initial point. Global solutions in nonconvex design spaces may be missed. DAKOTA's `optpp_g_newton` and `optpp_bcg_newton` methods differ in their support for bound constraints. Since bound constraints are commonly very important for keeping parameters within physically meaningful ranges, `optpp_bcg_newton` will often be the method of choice for parameter estimation.

Neither `optpp_g_newton` nor `optpp_bcg_newton` support general linear or nonlinear constraints. If these types of constraints are present (fairly rare in typical estimation problems), general-purpose optimization methods such as those available in the DOT and NPSOL libraries can be used (see DOT Library on page 54 and NPSOL Library on page 55). While neither DOT nor NPSOL exploit the special structure of a sum of the squares objective function, both are effective general-purpose algorithms for solving constrained minimization problems.

DAKOTA manages the following inputs for the Gauss-Newton method from its input specification: the maximum number of iterations and function evaluations, the convergence tolerance, the output verbosity, the search method, the maximum step, and the gradient tolerance. See OPT++ Methods on page 165 for additional details on these specifications.

Parameter Study Capabilities

Introduction

Parameter study methods in the DAKOTA system involve the computation of response data sets at a selection of points in the parameter space. The response functions are not linked to any specific interpretation, so the `m` functions in the DAKOTA response data set which are being catalogued by the study can consist of any optimization, least squares, or generic response function definition which is allowable by the responses input specification (see Responses Commands on page 141). This allows a parameter study iterator to be used in direct conjunction with optimization, least squares, and uncertainty quantification iterators without significant modification to the input file. In addition, response data sets are not restricted to function values only; gradients and Hessians of the response functions can also be catalogued by the parameter study. This allows for several different levels of "sensitivity analysis": (1) the variation of function values over parameter ranges provides indirect information on the sensitivity of the functions to those parameters, (2) derivative information can be computed numerically, provided analytically by the simulator, or both (mixed gradients) in directly determining sensitivity information at a point or points in parameter space, and (3) the variation of derivative quantities through the parameter space can be investigated.

In addition to the cited sensitivity analysis applications, parameter study capabilities are also commonly used for investigating simulation nonsmoothness issues (so that models can be tuned for use with gradient-based optimization algorithms), generating parameter and response ensembles for response surface generation or parameter space visualization, and performing code verification (verifying simulation robustness) through parameter ranges of interest. A parameter study iterator can also be used as either a pre-processor (to identify a good starting point) or a post-processor (for post-optimality analysis) within a multilevel hybrid optimization strategy (see Multilevel Hybrid Optimization on page 71), since each parameter study iterator can accept the best design point found in a previous study as its starting point or pass along its best design point for subsequent iteration or both. Note that only those parameter studies which use initial values (see Initial Values on page 63) will be affected by accepting the best design point from previous iteration. The best design point found in a parameter study is defined to be the point with the least constraint violation, or if there are no violations, the point with the lowest objective function.

Parameter study iterators will iterate any set of variables (any combination of design, uncertain, and state variables) into any set of responses (any function, gradient, and Hessian definition), so there are no restrictions on valid data set definitions. More specifically, parameter study iterators draw no distinction between different types of variables and different types of response functions. They simply pass all of the variables defined in the variables specification into the interface, from which they expect to retrieve all of the responses defined in the responses specification. The only subtle distinction involves the set of variables for which function

derivatives are computed. When gradient and/or Hessian information is being catalogued in the parameter study, then it is assumed that derivative components will be computed with respect to all of the *continuous* variables (continuous design, uncertain, and state variables) specified. The omission of discrete variables from gradient vectors and Hessian matrices is common among all iterators (since derivatives with respect to discrete variables do not exist); however, inclusion of all continuous variables differs from optimization and least squares iterators (which assume derivatives only with respect to the continuous design variables) and from nondeterministic analysis iterators (which assume derivatives only with respect to the uncertain variables). Lastly, while discrete variables (if present) will be mapped through the interface, enumeration of the discrete values of these variables by the parameter study methods is not yet supported.

Initial Values

The vector and centered parameter studies use the initial values of the variables from the variables commands specification (see Variables Commands on page 134) as the starting point and the central point of the parameter studies, respectively. In the case of design variables, the `initial_point` is used. In the case of state variables, the `initial_state` is used. In the case of uncertain variables, there is no initial value specification and 0.0 is used initially for each of these variables (NOTE: the mean might be a better value than 0.0). Therefore, in the following discussions, "Initial Values" are defined by `initial_point`, `initial_state`, and 0.0 for the design, state, and uncertain variables specified in the study, respectively.

Data Cataloguing

All parameter study algorithms catalogue the parameters and responses for each function evaluation in a special file named `dakota_pstudy.dat`. This file is intended to simplify plotting of parameter study data by making the data available in concise form separate from the other information available in the main output file (i.e., `dakota.out`).

Vector Parameter Study

The vector parameter study computes response data sets at selected intervals along a one-dimensional vector in parameter space. This capability encompasses both single-coordinate parameter studies (to study the effect of a single variable on a response set) as well as multiple coordinate vector studies (to investigate the response variations along some n-dimensional vector). In addition to these uses, this capability is used recursively within the implementations of the centered and multidimensional parameter studies (see Centered Parameter Study on page 66 and Multidimensional Parameter Study on page 67).

Dakota's vector parameter study includes three possible specification formulations which are used in conjunction with the Initial Values to define the vector and steps of the parameter study:

```
{final_point = <LISTof><REAL>} and {step_length = <REAL>}
```

```
{final_point = <LISTof><REAL>} and {num_steps = <INTEGER>}
{step_vector = <LISTof><REAL>} and {num_steps = <INTEGER>}
```

In each of these three cases, the Initial Values are used as the parameter study starting point and the specification selected from the three above defines the orientation and length of the vector as well as the increments to be evaluated along the vector. Several examples starting from Initial Values of 1.0, 1.0, 1.0 are included below:

`final_point` = 1.0, 2.0, 1.0 and `step_length` = .4:

```
Parameters for function evaluation 1:
                        1.0000000000e+00 d1
                        1.0000000000e+00 d2
                        1.0000000000e+00 d3
Parameters for function evaluation 2:
                        1.0000000000e+00 d1
                        1.4000000000e+00 d2
                        1.0000000000e+00 d3
Parameters for function evaluation 3:
                        1.0000000000e+00 d1
                        1.8000000000e+00 d2
                        1.0000000000e+00 d3
```

`final_point` = 2.0, 2.0, 2.0 and `step_length` = .4:

```
Parameters for function evaluation 1:
                        1.0000000000e+00 d1
                        1.0000000000e+00 d2
                        1.0000000000e+00 d3
Parameters for function evaluation 2:
                        1.2309401077e+00 d1
                        1.2309401077e+00 d2
                        1.2309401077e+00 d3
Parameters for function evaluation 3:
                        1.4618802154e+00 d1
                        1.4618802154e+00 d2
                        1.4618802154e+00 d3
Parameters for function evaluation 4:
                        1.6928203230e+00 d1
                        1.6928203230e+00 d2
                        1.6928203230e+00 d3
Parameters for function evaluation 5:
                        1.9237604307e+00 d1
                        1.9237604307e+00 d2
                        1.9237604307e+00 d3
```

`final_point` = 2.0, 2.0, 2.0 and `num_steps` = 4:

```
Parameters for function evaluation 1:
                        1.0000000000e+00 d1
                        1.0000000000e+00 d2
                        1.0000000000e+00 d3
Parameters for function evaluation 2:
                        1.2500000000e+00 d1
                        1.2500000000e+00 d2
                        1.2500000000e+00 d3
```

```
     Parameters for function evaluation 3:
                          1.50000000000e+00 d1
                          1.50000000000e+00 d2
                          1.50000000000e+00 d3
     Parameters for function evaluation 4:
                          1.75000000000e+00 d1
                          1.75000000000e+00 d2
                          1.75000000000e+00 d3
     Parameters for function evaluation 5:
                          2.00000000000e+00 d1
                          2.00000000000e+00 d2
                          2.00000000000e+00 d3
```

`step_vector` = .1, .1, .1 and `num_steps` = 4:

```
     Parameters for function evaluation 1:
                          1.00000000000e+00 d1
                          1.00000000000e+00 d2
                          1.00000000000e+00 d3
     Parameters for function evaluation 2:
                          1.10000000000e+00 d1
                          1.10000000000e+00 d2
                          1.10000000000e+00 d3
     Parameters for function evaluation 3:
                          1.20000000000e+00 d1
                          1.20000000000e+00 d2
                          1.20000000000e+00 d3
     Parameters for function evaluation 4:
                          1.30000000000e+00 d1
                          1.30000000000e+00 d2
                          1.30000000000e+00 d3
     Parameters for function evaluation 5:
                          1.40000000000e+00 d1
                          1.40000000000e+00 d2
                          1.40000000000e+00 d3
```

For additional information, refer to the commands specification for Vector Parameter Study on page 176.

List Parameter Study

The list parameter study computes response data sets at selected points in parameter space. These points are explicitly specified by the user and are not confined to lie on any line or surface.

This iterator requires the following specification:

```
     {list_of_points = <LISTof><REAL>}
```

This parameter study simply performs simulations for the first parameter set (the first `n` entries in the list), followed by the next parameter set (the next `n` entries), and so on, until the list of points has been exhausted. Since the Initial Values will not be used, they need not be specified.

An example specification which would result in the same parameter sets as in the first example in Vector Parameter Study on page 63 would be:

```
list_of_points = 1.0, 1.0, 1.0, 1.0, 1.4, 1.0, 1.0, 1.8, 1.0
```

For additional information, refer to the commands specification for List Parameter Study on page 178.

Centered Parameter Study

The centered parameter study executes multiple vector parameter studies, one per parameter, centered about the specified Initial Values. This is useful for investigation of function contours in the vicinity of a specific point. For example, after computing an optimum design, this capability could be used for post-optimality analysis in verifying that the computed solution is actually at a minimum or constraint boundary and in investigating the shape of this minimum or constraint boundary.

This iterator requires the following specifications:

```
{percent_delta = <REAL>}
{deltas_per_variable = <INTEGER>}
```

where `percent_delta` specifies the size of the increments in percent and `deltas_per_variable` specifies the number of increments per variable in each of the plus and minus directions.

For example, with Initial Values of 1.0, 1.0, `percent_delta` = 10.0, and `deltas_per_variable` = 2, five function evaluations (two minus deltas, the center point, and two plus deltas) would be performed per variable:

```
Parameters for function evaluation 1:
                       8.0000000000e-01 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 2:
                       9.0000000000e-01 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 3:
                       1.0000000000e+00 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 4:
                       1.1000000000e+00 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 5:
                       1.2000000000e+00 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 6:
                       1.0000000000e+00 d1
                       8.0000000000e-01 d2
Parameters for function evaluation 7:
                       1.0000000000e+00 d1
                       9.0000000000e-01 d2
Parameters for function evaluation 8:
```

```
                        1.0000000000e+00 d1
                        1.0000000000e+00 d2
     Parameters for function evaluation 9:
                        1.0000000000e+00 d1
                        1.1000000000e+00 d2
     Parameters for function evaluation 10:
                        1.0000000000e+00 d1
                        1.2000000000e+00 d2
```

This set of points in parameter space is depicted in Figure 13

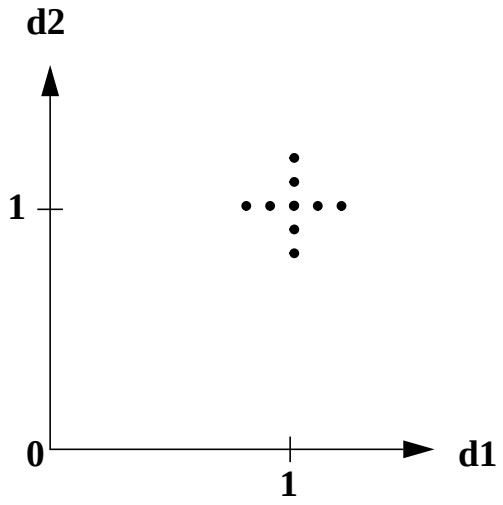


Figure 13 Example centered parameter study.

For additional information, refer to the commands specification for Centered Parameter Study on page 178.

Multidimensional Parameter Study

The multidimensional parameter study computes response data sets for an n-dimensional hypergrid of points. Each continuous variable is partitioned into equally spaced intervals between its upper and lower bounds, and each combination of the values defined by these partitions is evaluated. The number of function evaluations performed in the study is:

$$\prod_{i=1}^{n} (\texttt{partitions}_i + 1) \tag{6}$$

The partitions information is specified as follows:

```
{partitions = <LISTof><INTEGER>}
```

where the entries in the list specify the number of partitions for each continuous variable (i.e., $\texttt{partitions}_i$). Since the Initial Values will not be used, they need not be specified.

In a two variable example problem with d1 ∈ [0,2] and d2 ∈ [0,3] (as defined by the upper and lower bounds specified in the variables specification) and with `partitions = 2,3`, the interval [0,2] is divided into two equal-sized partitions and the interval [0,3] is divided into three equal-sized partitions. This two-dimensional grid, shown in Figure 14,

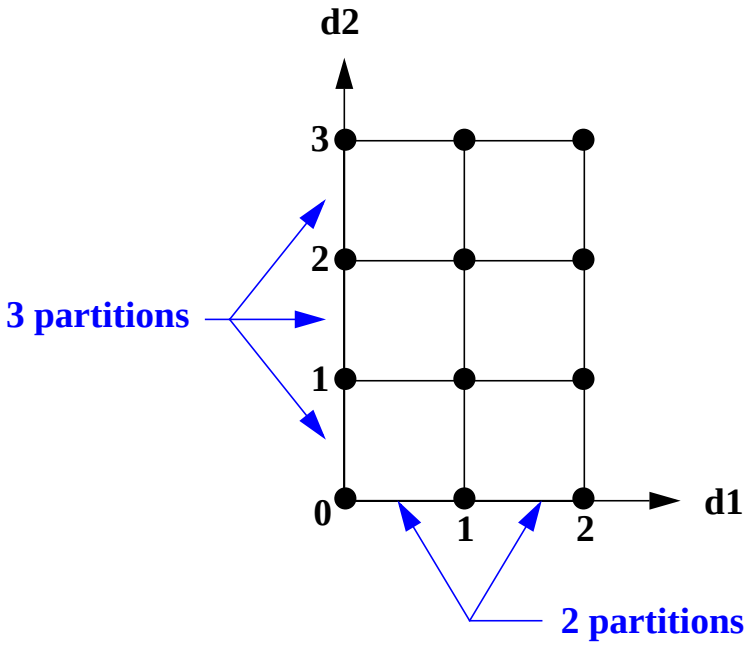


Figure 14 Example multidimensional parameter study

would result in the following twelve function evaluations:

```
Parameters for function evaluation 1:
                       0.0000000000e+00 d1
                       0.0000000000e+00 d2
Parameters for function evaluation 2:
                       1.0000000000e+00 d1
                       0.0000000000e+00 d2
Parameters for function evaluation 3:
                       2.0000000000e+00 d1
                       0.0000000000e+00 d2
Parameters for function evaluation 4:
                       0.0000000000e+00 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 5:
                       1.0000000000e+00 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 6:
                       2.0000000000e+00 d1
                       1.0000000000e+00 d2
Parameters for function evaluation 7:
                       0.0000000000e+00 d1
```

```
                        2.0000000000e+00 d2
Parameters for function evaluation 8:
                        1.0000000000e+00 d1
                        2.0000000000e+00 d2
Parameters for function evaluation 9:
                        2.0000000000e+00 d1
                        2.0000000000e+00 d2
Parameters for function evaluation 10:
                        0.0000000000e+00 d1
                        3.0000000000e+00 d2
Parameters for function evaluation 11:
                        1.0000000000e+00 d1
                        3.0000000000e+00 d2
Parameters for function evaluation 12:
                        2.0000000000e+00 d1
                        3.0000000000e+00 d2
```

For additional information, refer to the commands specification for Multidimensional Parameter Study on page 179.

Strategy Capabilities

Multilevel Hybrid Optimization on page 71 The Coupled Approach on page 73

Introduction

Dakota's strategy layer was developed to provide a means for management of multiple iterators, models, and approximations. It was driven by the observed need for high level "meta-control" of optimization and other system analysis processes. By providing an additional level of logic on top of the iterators, it becomes possible to develop adaptive strategies which switch and refine iterators and models based on run-time performance assessments. This adaptive control can lead to automated procedures which exploit the capabilities of several iterators, manage varying model fidelity, and incorporate approximations for the purpose of navigating to the solution more reliably and efficiently than with single method approaches.

Several advanced approaches are available within the strategy class hierarchy shown in Figure 12. In the multilevel hybrid strategy, two or more optimizers are combined in a hybrid strategy in which the best point from one iterator is used as the starting point for a subsequent iterator. Fine-grained control, effective switching metrics, and the existence of multiple iteration follow-on candidates from some global methods are important research issues. The single method strategy invokes only one iterator and can be viewed as a "fall through" strategy in that no additional coordination is performed at the strategy layer and control falls through to the iterator. The branch and bound strategy is used for solution of mixed continuous/discrete applications. The nondeterministic optimization strategy (a.k.a. optimization under uncertainty) incorporates an uncertainty quantification within the optimization process. It can be used to minimize stochastic quantities, such as probability of failure. Use of nested and segregated frameworks is an important research issue. In the sequential approximate optimization strategy, an optimizer is interfaced with an approximate design space representation in order to find an approximate optimal solution. "Exact" evaluations at this approximate optimal solution are then used to update the approximation and restart the sequence. Here, the effective use of experimental design techniques, the development of accurate approximations using a minimal number of function evaluations, and the development of provably convergent approaches for sequential approximation are important research issues.

In addition to management of multiple iterators and models, the strategy layer implements the master-slave algorithm for exploiting parallelism by providing separation of iterator code (the master processor) from model server code (the slave processors). Refer to Exploiting Parallelism on page 99 for additional details.

Several strategies continue to be works in progress. Therefore, "STATUS" statements have been added at the end of each of the following strategy descriptions.

Single Method

The single method strategy is implemented within the **SingleMethodStrategy** class and is invoked with the `single_method` selection in the user's strategy section specification (see Single Method Commands on page 152 for additional specification details). The single method strategy is also used as the default strategy if no strategy specification is included in the user's input file.

The single method strategy is used to invoke a single **DakotaIterator** object which iterates on a single **DakotaModel** object. This "strategy" is provided since the main program of DAKOTA is bound to the instantiation and execution of one of the strategies within the **DakotaStrategy** class hierarchy. That is, even if coordination of multiple iterators and models is not needed, a simple strategy is still required to create the iterator and the model and perform the iteration.

STATUS: Fully operational.

Multilevel Hybrid Optimization

The multilevel hybrid strategy is implemented within the **MultilevelOptStrategy** class and is invoked with the `multi_level` selection in the user's strategy section specification (see Multilevel Hybrid Optimization Commands on page 152 for additional specification details). There are three multilevel approaches available: the uncoupled approach, the uncoupled adaptive approach, and the coupled approach.

The Uncoupled Approach

In the uncoupled approach, a sequence of methods is invoked in the order specified in a method list specification. The best solution from each method is used as the starting point for the following method. Method switching is governed by the separate convergence controls of each method; that is, *each iterator is allowed to run to its own internal definition of completion without interference*. Individual method completion may be determined by convergence criteria (e.g., `convergence_tolerance`) or iteration limits (e.g., `max_iterations`).

The basic algorithm, in simplified form, is shown in Figure 15:

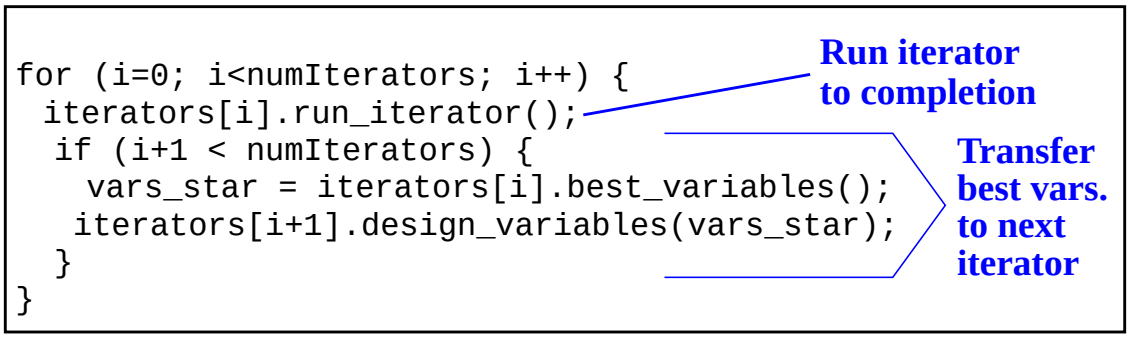


Figure 15 Uncoupled multilevel hybrid optimization strategy

where `run_iterator()` and `best_variables()` are virtual functions which define a generic behavior valid for all iterators for which the specific implementation can vary. This strategy is relatively simple since the only coordination required is the transferral of the best solution between successive iterators.

STATUS: Fully operational.

The Uncoupled Adaptive Approach

The simple uncoupled approach is being extended through development of more finely grained iterator control using “iterator++” overloaded operators. In this approach, *optimization algorithms are incremented one optimization cycle at a time* and intermediate performance data are returned as a basis for adaptive switching. For example, a gradient-based optimization cycle consists of computing objective and constraint gradients, computing a search direction using these gradients, and performing a line search along the search direction to find an improved point. By executing an optimizer one cycle at a time, a history of improved points can be logged and relative performance metrics can be defined. These performance metrics are fundamentally different than the convergence metrics used in the nonadaptive approach: convergence metrics typically assess whether the method can make any additional progress within a specified tolerance (e.g., are the Kuhn-Tucker conditions for a constrained minimum approximately satisfied?) whereas performance metrics measure the rate of progress (i.e., has the rate of improvement in objective minimization and/or constaint satisfaction decreased significantly?). While this distinction is somewhat fuzzy since some convergence metrics (e.g., convergence tolerance on relative change in the objective function) are similar to a rate of progress metric, the key point is that we may want to terminate a method prior to its formal convergence and switch to another method. Put another way, this distinction can be cast as “are we there?” versus “how fast are we getting there?” Certainly, the former question is most appropriate when one method is available; however, the availability of multiple methods in a hybrid strategy admits a more aggressive approach.

The basic algorithm, in simplified form, is shown in Figure 16:

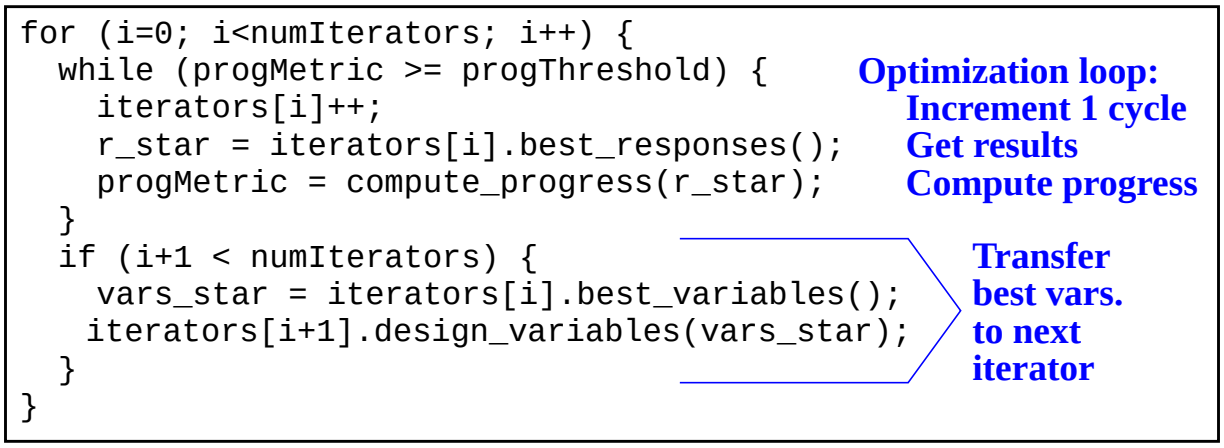


Figure 16 Uncoupled adaptive multilevel hybrid optimization strategy

where the overloaded ++ operator, `best_responses()`, and `best_variables()` are virtual functions, and `progThreshold` contains a user specified progress threshold (see Multilevel Hybrid Optimization Commands on page 152). This strategy requires considerably more sophistication than the standard uncoupled approach since additional mechanisms for cycle control and progress computation are required for all of the optimizers.

Definition of an appropriate progress metric can be troublesome when attempting to encompass broad classes of methods. In general, the DAKOTA approach to this is to compute rate of convergence history information over a series of optimization cycles. When rate of improvement slows from previous cycles, the `progMetric` (normalized between 0.0 and 1.0) will be small and may fall below the `progThreshold` and trigger a method switch. By selecting a large `progThreshold` value (closer to 1.0), the user can specify aggressive method switching in which a slight decrease in convergence rate will trigger a switch, whereas a small `progThreshold` (closer to 0.0) will be considerably more tolerant of (perhaps transient) decreases in convergence rate. In this latter case, the adaptive approach may perform much like the uncoupled approach and, in fact, the internal convergence criteria may trigger method completion prior to `progMetric` triggering a method switch.

STATUS: adaptive “iterator++” approach under development.

The Coupled Approach

The coupled approach implements specific hybrid algorithms available within SGOPT which exploit a tighter coupling to achieve peak performance. For example, whereas an uncoupled GA/local search hybrid would use the best solution found from a GA to start a local search, a coupled hybrid would use local search to occasionally improve members in an evolving GA population. That is, in an uncoupled approach, multiple methods run one at a time sharing only their best results at completion, while in a coupled approach, methods are working together throughout the strategy to synergistically improve the solution.

Whereas in the uncoupled approach, the number of methods and possible combinations are unlimited, the coupled approach has only a few allowable method combinations. Only two methods are specified (as opposed to an open-ended method list): one global method and one local method. The allowable global methods are currently `sgopt_pga_real` and `sgopt_strat_mc`, and the allowable local methods are currently `sgopt_solis_wets`, `sgopt_coord_ps`, and `sgopt_coord_sps`. More methods will be allowable selections in future releases. In the `sgopt_pga_real` case, local search is used to periodically improve GA population members. In the `sgopt_strat_mc` case (also known as "multi-start local search"), local search is applied with a prescribed probability to Monte Carlo samples. When a local search is performed, it is performed immediately (prior to evaluation of the next sample). This type of iterator coordination makes it a coupled approach by definition, although in this case it only differs from an uncoupled approach (in which local searches would be performed after all sampling was complete) in the effect of order-dependent termination criteria such as `max_function_evaluations` and, possibly, in how iteration follow-on candidates are selected. The `sgopt_strat_mc` coupled hybrid is not a particularly sophisticated hybrid and is not recommended for optimization with expensive engineering simulations. It is primarily useful for its theoretical simplicity as a benchmark for comparison with more efficient approaches (i.e., the GA coupled hybrids).

STATUS: strategy wrapper for SGOPT multi-start and global/local hybrids under development.

Sequential Approximate Optimization

The sequential approximate optimization strategy is implemented within the **SeqApproxOptStrategy** class and is invoked with the `seq_approximate_opt` selection in the user's strategy section specification (see Sequential Approximate Optimization Commands on page 154 for additional specification details).

In the `seq_approximate_opt` strategy, two models (`actualModel` and `approxModel`) and one iterator (`selectedIterator`) are constructed. The `approxModel` contains one of the approximation methods from the hierarchy described in The Approximation Interface on page 95 and the `actualModel` contains one of the simulation interfacing methods described in The Application Interface on page 79. First, the approximation within `approxModel` is built using function evaluations which are selected via a design of experiments and which are performed with the `actualModel`. The `selectedIterator` then iterates on `approxModel` (it is bound to this model in the strategy constructor) and computes an approximate optimum. This approximate optimum is evaluated with the `actualModel` and the resulting parameter/response pair is evaluated for improvement from the previous cycle and for convergence of the process. Based on the observed improvement, the extent (i.e. bounds) of the approximation is modified via trust region concepts. If the process is not converged, then the new parameter/response pair from the `actualModel` is used to update the `approxModel`. Iteration is then reinitiated on the updated `approxModel` and the process repeats until convergence. It is worth emphasizing that the iterator only iterates on `approxModel`. The

`actualModel` is only used for building and updating the approximation and is never iterated directly.

The basic algorithm, in simplified form, is shown in Figure 17:

```
approxModel.build_approximation();                          Initialize approx.
while (conv_metric > conv_tol) {                            Main loop:
  selectedIterator.run_iterator();                              Optimize approx.
  v_star = selectedIterator.best_variables();                   Get approx. soln.
  r_star = actualModel.compute_response(v_star);                Evaluate soln.
  approxModel.modify_approximation(r_star);                     Modify extent
  approxModel.update_approximation(v_star,r_star);              Add new data
}
```

Figure 17 Sequential approximate optimization strategy

where `run_iterator()` and `best_variables()` are virtual functions within the iterator hierarchy and `build_approximation()`, `modify_approximation()`, and `update_approximation()` are virtual functions within the interface hierarchy. It is critical for the `modify_approximation()` step to perform operations (e.g., modify trust regions) which assure convergence of the sequential process.

STATUS: Operational, but undergoing convergence enhancements.

Optimization Under Uncertainty

The optimization under uncertainty strategy is implemented within the **NonDOptStrategy** class and is invoked with the `opt_under_uncertainty` selection in the user's strategy section specification (see Optimization Under Uncertainty Commands on page 154 for additional specification details).

In the `opt_under_uncertainty` strategy, two models (`designModel` and `uncertainModel`) and two iterators (`optIterator` and `nonDIterator`) are constructed. The `designModel` provides a mapping of a set of design variables into a set of design responses (an objective function and constraints) through the use of one interface, whereas the `uncertainModel` maps a set of uncertain variables into a set of uncertain responses through another interface. The `optIterator` iterates on `designModel` in the optimization loop and the `nonDIterator` iterates on `uncertainModel` in the uncertainty quantification loop. Note that *the mappings for both models are deterministic*; it is the ensemble of `uncertainModel` mappings based on the set of uncertain variable realizations that provide the desired statistics for the uncertain responses.

In the case of a nested approach, the optimization loop is the outer loop which seeks to optimize a nondeterministic quantity (e.g., minimize probability of failure). The uncertainty quantification

inner loop evaluates this nondeterministic quantity (e.g., compute the probability of failure) on each optimization function evaluation.

For a segregated approach, the loops are not nested, rather they are executed in repeated succession until convergence. The coupling of the uncertainty quantification to the design process occurs through the adjustment of the optimization objective and constraints in order to modify the statistical performance of the optimal design computed (e.g., to adjust the probability of failure of a minimum weight design by changing the stress allowables). The nested approach is desirable since it removes the compounded expense of nested loops; however, the logic for modifying the design objectives is heuristic and application-dependent.

STATUS: Under development. Not yet operational.

Branch and Bound

The branch and bound strategy is implemented within the **BranchBndStrategy** class and is invoked with the `branch_and_bound` selection in the user's strategy section specification (see Branch and Bound Commands on page 155 for additional specification details).

It employs the PICO branching engine ([Eckstein, J., Hart, W.E., and Phillips, C.A., 1997]) in combination with DAKOTA's multilevel parallelism facilities ([Eldred, M.S., and Schimel, B.D., 1999]) to enable parallel solution of nonlinear mixed continuous and discrete problems through parameter domain decomposition (branching) and nonlinear solution of optimization subproblems with relaxation of integrality constraints (bounding).

STATUS: Operational. To be available in DAKOTA V1.2.

Simulation Interfacing

Dakota Interface Abstraction

DAKOTA's interfacing capabilities are encompassed within an interface abstraction. This abstraction is the general concept of mapping a set of parameters into a set of responses for the purpose of performing a function evaluation. The implementation of this abstraction within the **DakotaInterface** class hierarchy involves the use of a variety of evaluation mechanisms and communication protocols, each of which shares this common functionality of parameter to response mapping. Supported evaluation mechanisms currently include interfacing with simulation codes, employing response approximations, and employing internal testing functions. And currently supported communication protocols include system calls with file communication, direct function invocations with parameter list communication, and parallel message-passing (for either direct communication with simulations or in combination with system call and direct function invocation and communication). In addition, coordination of disciplinary simulations for multidisciplinary optimization with the global sensitivity equations is a natural extension to the supported evaluation mechanisms, and CORBA and JAVA binding with geographically distributed analysis services (e.g., for interface with Sandia's CORBA-based Product Realization Environment) is an attractive extension to the supported communication protocols. These additions will continue to extend the breadth of possible DAKOTA problem solving environments.

DAKOTA provides a framework for the implementation of these evaluation mechanisms and communication protocols within the **DakotaInterface** class hierarchy shown in Figure 18. The **DakotaInterface** base class provides the starting point from which specialized interface mechanisms are created. This base class contains the virtual `map` function which each derived class must redefine in order to implement its particular mechanism for generating responses from a set of parameters. Furthermore, this base class provides the envelope for derived letter classes in a letter/envelope idiom design. The letter/envelope idiom is an advanced C++ construct which provides mechanisms for enhanced polymorphism (the envelope is a generic handle for any derived class) and for smart memory management through reference counting [Coplien, J.O., 1992].

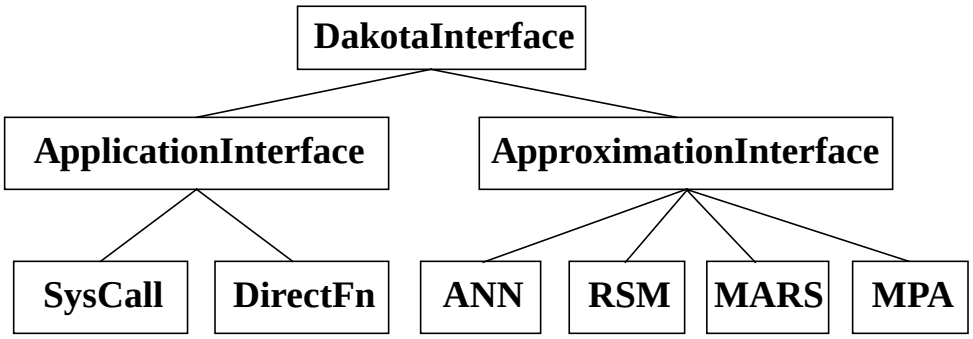


Figure 18 The DakotaInterface class hierarchy

The **ApplicationInterface** and **ApproximationInterface** classes provide base classes for those interfaces dealing with simulation codes and response approximations, respectively. Within the **ApplicationInterface** branch, simulation codes may be interfaced using system calls (the **SysCallApplicInterface** class) or through direct function calls (the **DirectFnApplicInterface** class). The system call application interface communicates with the simulation it spawns through the use of files. In this case, data formats are very important (see DAKOTA File Data Formats on page 85). However, in the direct function application interface case, C++ references to data structures are passed directly to the simulation; files and specialized data formats are not needed. In addition to invoking simulations which are linked into the DAKOTA executable, the direct function application interface is also used for algorithm testing with internal test functions, so it serves a dual purpose.

The **ApproximationInterface** branch implements a variety of approximations which can be used as surrogates in place of actual simulations. The **ANNApproxInterface**, **RSMApproxInterface**, and **MARSApproxInterface** classes implement artificial neural networks, response surface methods, and multivariate adaptive regression splines, respectively. In addition, an **MPAApproxInterface** class is planned for implementing multipoint approximations. Each of these approximation classes must implement methods for building, updating, modifying, and performing function evaluations with the approximation.

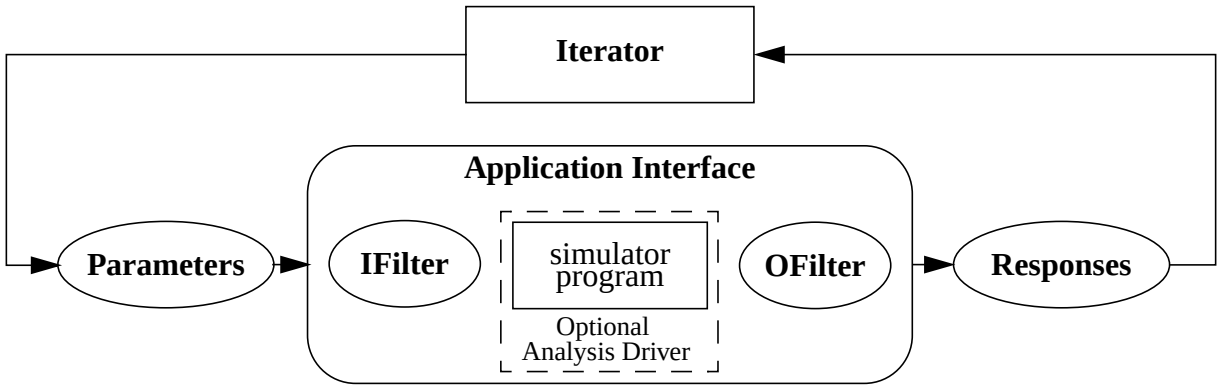


Figure 19 The Application Interface Concept

By providing a generic interface for the mapping of a set of parameters (e.g., the vector of design variables) into a set of responses (e.g., an objective function, constraints, and sensitivities), the Application Interface hides the specific complexities of a given problem from the iterator method. All of an application's disciplinary specifics and implementation details are encapsulated within the Application Interface box in Figure 19. External to that box, the data flows between the iterator and the simulator are generic and abstract. Isolation of complexity through the development of generic interfaces is a cornerstone of object-oriented design (the concept of "one interface, many methods").

Housed within the Application Interface are three main components. The input filter program ("IFilter" in Figure 19) provides a communication link which transforms the set of DAKOTA input parameters into the input required by the simulator program. The simulator program reads its input and computes its results (a driver program/script is optional and is used to accomplish nontrivial command syntax and/or progress monitoring). Finally, the output filter program ("OFilter" in Figure 19) provides another communication link through the recovery of data from the simulation results and the computation of the desired response data set. The two filter programs are generally application specific, although it is a project goal to maximize reusability through the build-up of generic libraries of filtering capabilities over time. Note that the input and output filters are part of the Application Interface and are named "input" and "output" relative to the simulator program.

The Application Interface mapping can be accomplished in several ways. The two ways currently in use are the direct function and system call methods. The former uses direct invocation of linked-in functions to perform the parameter to response mapping, whereas the latter uses system calls to external programs and file-based communication to perform the mapping. In both of these cases, either a 3-piece interface or a 1-piece interface may be used, which differ in whether or not they use filter programs. The following sections describe these two approaches as embodied in the direct function application interface and system call application interface classes.

Following the discussion of the direct function and system call application interfaces, techniques for capturing simulation failures within application interfaces are presented. Failure recovery options include abort, retry, recover, and continuation.

The Direct Function Application Interface

The direct function application interface capability may be used to invoke simulation codes which are linked into the DAKOTA executable or to invoke internal test functions for algorithm performance testing. This option, in an earlier incarnation, was used in the TWAFER CVD heater design application ([Moen, C.D., Spence, P.A., Meza, J.C., and Plantenga, T.D., 1996], [Moen, C.D., Spence, P.A., and Meza, J.C., 1995], and [Meza, J.C., and Plantenga, T.D., 1995]) in order to improve data precision and efficiency by eliminating system calls for filter programs and file transfer of parameter/response data. In this earlier incarnation, a system call was still required for the simulator program since, although the TWAFER filters were compiled into the Dakota executable, the TWAFER simulation code was not. In the current direct function and system call capabilities, the entire parameter to response mapping must be accomplished with either system calls or direct function calls. No combinations are allowed.

In order to use the direct function capability with a new simulation or new test function (not previously interfaced), the following steps have to be performed:

1. the functions to be invoked must have their main programs changed into callable functions with the following prototype: `int function_name(const DakotaVariables& vars, const DakotaIntArray& asv, DakotaResponse& response)`. The same prototype is used for filter and analysis programs (which departs from the distinctions between filters and analysis shown in the command line file name passing procedures of The System Call Application Interface on page 81).
2. the if-else blocks in **DirectFnApplicInterface::execute()** must be extended to include the new function names with the proper prototypes
3. the DAKOTA system must be recompiled and linked with the new function object files or libraries

Various header files will have to be included in order to compile successfully, both within the **DirectFnApplicInterface** class (in order for the class to recognize the new functions) and within the new functions themselves (in order to recognize the `DakotaVariables`, `DakotaIntArray`, and `DakotaResponse` types).

The direct function capability is new and evolving. Future work may include removal of the dependence of user-supplied routines on DAKOTA objects by replacing the objects with more fundamental data structures (vectors of ints and doubles), and installation of the “builder pattern” (see [Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995]) for management of multiple user-supplied routines.

3-piece Interface

In the 3-piece case, the parameters to responses mapping occurs in 3 separate steps. Each of the functions identified by the `input_filter`, `analysis_driver`, and `output_filter` specifications will be invoked in succession.

1-piece Interface

If the `analysis_driver` specified in the interface section is to perform the complete parameters to responses mapping and no additional filters are needed, then only one function invocation will occur. This 1-piece interface is accomplished through the use of the "NO_FILTER" option (the default) in the `input_filter` and `output_filter` specifications.

The System Call Application Interface

The system call approach invokes a simulation code or simulation driver by using the `system` function from the C standard library ([Kernighan, B.W., and Ritchie, D.M., 1988]) to create a new process. This new process communicates with DAKOTA through parameter and response files. The system call approach eliminates the need to modify simulation source code since the simulation can be initiated via its standard invocation procedure and then coordinated with any variety of tools for pre- and post-processing. The simulation can be viewed as a "black box" for which the filter programs provide the communication links and the parameters and responses files provide the communication data. This approach has been widely used in [Eldred, M.S., Hart, W.E., Bohnhoff, W.J., Romero, V.J., Hutchinson, S.A., and Salinger, A.G., 1996], [Eldred, M.S., Outka, D.E., Bohnhoff, W.J., Witkowski, W.R., Romero, V.J., Ponslet, E.R., and Chen, K.S., 1996], and many others. The system call approach involves more process overhead than the direct function approach; however, this is most often of very little significance compared to the expense of the simulations. Lastly, the system call approach can suffer from precision problems if care is not taken to preserve data precision in parameter and response file I/O. The following sections describe system call functionality for the cases of separate filter programs (the 3-piece interface) and no filter programs (the 1-piece interface).

3-piece Interface

The syntax of the system call that Dakota performs for a 3-piece interface is

```
(ifilter_name params.in; analysis_driver_name; ofilter_name
     results.out)
```

in which the input filter, analysis, and output filter processes are combined into a single system call through the use of semi-colons and parentheses (see [Anderson, G., and Anderson, P., 1986]). This single system call is equivalent to 3 separate system calls; however, they are bound

together to simplify asynchronous process management (test and receive synchronization operations).

The input filter is passed the name of the parameters file on the command line and the output filter is likewise passed the name of the results file on the command line. By passing the names of files on the command lines of executable programs, Dakota can communicate with these executables using unique and/or tagged file names (e.g., UNIX temporary files or root names tagged with function evaluation number). Having the option of using unique file names allows for multiple simultaneous simulations running in a common disk space.

1-piece Interface

If the `analysis_driver` specified in the interface section is to perform the complete parameters to responses mapping and no additional filters are needed, then only one process will appear in the system call. This 1-piece interface is accomplished through the use of the "NO_FILTER" option (the default) in the `input_filter` and `output_filter` specifications.

The system call syntax is:

```
(analysis_driver_name params.in results.out)
```

Since there are no filters, the names of the parameters and results files are both passed on the command line to the `analysis_driver`.

Additional Features

This section describes interfacing options for file saving, file tagging, Unix temporary files, and common filtering operations. For details on specification of these options, refer to Interface Commands on page 127. When executing DAKOTA, the actual system calls performed as well as informational messages on file renaming or removal are echoed to stdout in order for the user to verify proper operation of the software.

File saving

The `file_save` option in the interface specification allows the user to control whether parameters and results files are retained or removed from the working directory. Default behavior is to remove files once their use is complete in order to declutter working directories. However, by specifying `file_save` in the interface specification, these files will not be removed. This latter behavior is often useful for debugging communication between Dakota and simulator programs.

File tagging

The `file_tag` option in the interface specification allows the user to make the names of the parameters and results files unique by appending a function evaluation number to the root file

names specified in the `parameters_file` and `results_file` specifications. Default behavior is to not tag these files. The default behavior has the advantage of allowing the user to ignore command line argument passing and always read and write to/from the same file names, but has the disadvantage that nonunique file names may be overwritten from one function evaluation to the next. On the other hand, by specifying `file_tag` in the interface specification, these files become unique through the appended evaluation number. This is most often used when multiple simultaneous simulations are running in a common disk space, since it becomes necessary to prevent conflicts (file overwriting) between the simultaneous simulations by uniquely identifying files according to their evaluation number. *Special case:* When `file_save` is used without `file_tag`, untagged files are used in the function evaluation but are then moved to tagged file names after the function evaluation is complete (and before the next evaluation starts) in order to prevent overwriting files for which a `file_save` request has been given.

Unix temporary files

If `parameters_file` and `results_file` are not included in the interface specification, then the default mechanisms for file communication are Unix temporary files (e.g., `/usr/tmp/aaaa08861`). These files have unique names as created by the `tmpnam` utility from the C standard library ([Kernighan, B.W., and Ritchie, D.M., 1988]). This uniqueness makes it a requirement for the user's interface to retrieve the names of these files from the command line. File tagging is unnecessary with Unix temporary files (since they are already unique); thus, `file_tag` requests will be ignored. `file_save` requests will be honored, although this option is not recommended for the purpose of keeping the temporary file directory uncluttered.

Common filtering operations

A mechanism has been constructed for the implementation of common/generic filtering operations which are relatively application-independent. By providing mechanisms for common I/O filtering operations, the work in developing filters for new applications can be minimized. Examples of common filtering operations include design variable linking on the input filter side and filtering of noisy response time histories on the output filter side. These common filtering operations comprise a second level of filtering implemented externally to the inner layer of application-specific filtering. This additional filtering layer is encapsulated in the **ApplicationInterface** class and is currently inactive. That is, it is a placeholder for future extensions.

Examples

The NO_FILTER option

In a 1-piece interface (the NO_FILTER option), the user provides a single script or executable that accepts two command-line arguments: a parameters file name and a responses file name.

This executable must read the parameters file and write the appropriate data to the responses file. If a user creates a script/executable named “my_analysis” (the name of the `analysis_driver`), selects “params.in” as the `parameters_file` name and “results.out” as the `results_file` name, and employs the defaults of no file saving and no file tagging, then system calls with the following syntax will be spawned by Dakota:

```
(my_analysis params.in results.out)
```

If `file_tag` is requested, system calls like the following will be used:

```
(my_analysis params.in.1 results.out.1)
```

If UNIX temporary files are used (no `parameters_file` or `results_file` specification), system calls like the following will be used:

```
(my_analysis /usr/tmp/aaaa20305 /usr/tmp/baaa20305)
```

In the first of these three cases, the user need not retrieve the command line arguments since the same file names will be employed each time. With the latter two cases, the user must retrieve the command line arguments since the file names change on each evaluation. In the case of a C-shell script, the two command line arguments are retrieved using `$argv[1]` and `$argv[2]` (see [Anderson, G., and Anderson, P., 1986]). In the case of a C or C++ program, command line arguments are retrieved using `argc` (argument count) and `argv` (argument vector) [Kernighan, B.W., and Ritchie, D.M., 1988]. Fortran 77 does not support command line arguments; in this case, a shell script wrapper can be built around the Fortran program to handle unique file names (by, for example, creating a tagged working directory for the Fortran simulation and moving the unique file name to a hardwired file name within the working directory).

If `file_save` is not set, a file remove notification will follow the system call echo, e.g.:

```
Removing /usr/tmp/aaaa20305 and /usr/tmp/baaa20305
```

If nonunique file names are to be saved (`file_save` is set without either `file_tag` being set or UNIX temporary files being used), then these files will be saved by moving them to tagged files after the evaluation is complete to prevent overwriting them on subsequent evaluations. In this case, the following notification is echoed:

```
Files with nonunique names will be tagged to enable
     file_save:
Moving params.in to params.in.1
Moving results.out to results.out.1
```

The named filter option

In a 3-piece interface (the named filter option), the user chooses to create separate input and output filters that perform the data translations between Dakota and the simulator program. The input filter translates a standard Dakota parameters file into an analysis code input file, the simulator runs and produces data, and then the output filter translates the analysis code output file or database into a standard Dakota results file. If a user is employing an `analysis_driver` named “my_analysis,” an `input_filter` named “my_ifilter,” an `output_filter` named “my_ofilter,” selects “params.in” as the `parameters_file` name and “results.out” as the

`results_file` name, and employs the defaults of no file saving and no file tagging, then system calls with the following syntax will be spawned by Dakota:

```
(my_ifilter params.in; my_analysis; my_ofilter results.out)
```

If `file_tag` is requested, system calls like the following will be used:

```
(my_ifilter params.in.1; my_analysis; my_ofilter
     results.out.1)
```

If UNIX temporary files are used (no `parameters_file` or `results_file` specification), system calls like the following will be used:

```
(my_ifilter /usr/tmp/aaaa22490; my_analysis; my_ofilter /
     usr/tmp/baaa22490)
```

Similar to the 1-piece case, the user's input and output filters must retrieve the command line arguments in the latter two of the three cases above since the file names change on each evaluation. Identical to the 1-piece case, omitting the `file_save` flag will result in the following action

```
Removing /usr/tmp/aaaa22490 and /usr/tmp/baaa22490
```

and use of `file_save` with nonunique file names will result in actions of this type:

```
Files with nonunique names will be tagged to enable
     file_save:
Moving params.in to params.in.1
Moving results.out to results.out.1
```

DAKOTA File Data Formats

The central purpose of simulation interfaces is the mapping of a set of parameters into a set of responses. DAKOTA uses its own format for this data input/output within interfaces which employ *file transfer* of data (i.e., the system call application interface). Depending on the user's interface specification, DAKOTA will write the parameters file in either standard or APREPRO format. The latter option simplifies model parameterization using the APREPRO utility ([Sjaardema, G.D., 1992]). For the results file, only one format is supported.

Parameters file format (standard)

Prior to invoking an interface, DAKOTA creates a parameters file which contains the current parameter values and a set of function requests. This parameters file has the following standard format:

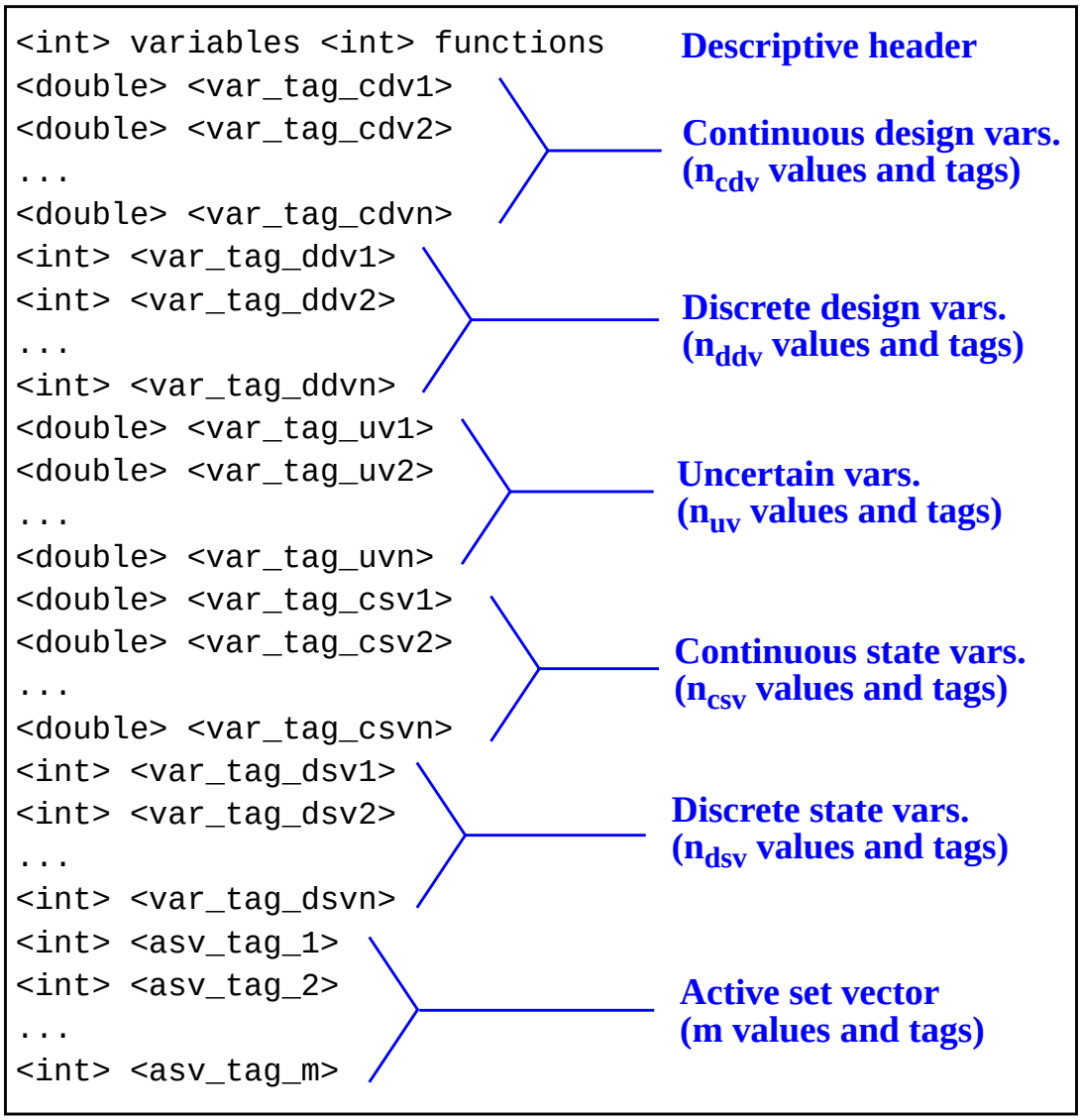


Figure 20 Parameters file data format, standard option

where "<int>" denotes an integer value, "<double>" denotes a double precision value, and "..." indicates omitted lines for brevity. The first line specifies the total number of variables (n) with its identifier string "variables" followed by the number of functions (m) with its identifier string "functions." These integers are useful for dynamic memory allocation within a simulator or filter program. The next n lines specify the current values and descriptors of all of the variables within the parameter set in the following order: continuous design, discrete design, uncertain, continuous state, and discrete state variables. The lengths of these vectors add to a total of n (that is, $n_{cdv}+n_{ddv}+n_{uv}+n_{csv}+n_{dsv}=n$). If any of the variable types are not present in the problem, then its block is omitted entirely from the parameters file. The descriptors are those specified in the user's Dakota input file, or if no descriptors have been specified, default descriptors are used. The next m lines specify the request vector for each of the m functions in the response data set. These integer codes indicate what data is required on the current function evaluation. Integer values of 0 through 7 denote a 3-bit binary representation of all possible

combinations of value, gradient, and Hessian requests for a particular function, with the most significant bit denoting the Hessian, the middle bit denoting the gradient, and the least significant bit denoting the value. The specific translations are shown in Table 8.

Table 8 Request vector codes

Integer Code	Binary representation	Meaning
7	111	Get Hessian, gradient, and value
6	110	Get Hessian and gradient
5	101	Get Hessian and value
4	100	Get Hessian
3	011	Get gradient and value
2	010	Get gradient
1	001	Get value
0	000	Get nothing, function is inactive

This request vector accomplishes two operations: (1) it manages the type of function data that is needed, and (2) it implements the active set strategy by providing a mechanism for distinguishing between active and inactive functions.

Parameters file format (APREPRO)

For the APREPRO format option, the same data is present and the same ordering is used as in the standard format. The difference is that numerical values are associated with their tags within `$$ { tag = value }` constructs as shown in Figure 21:

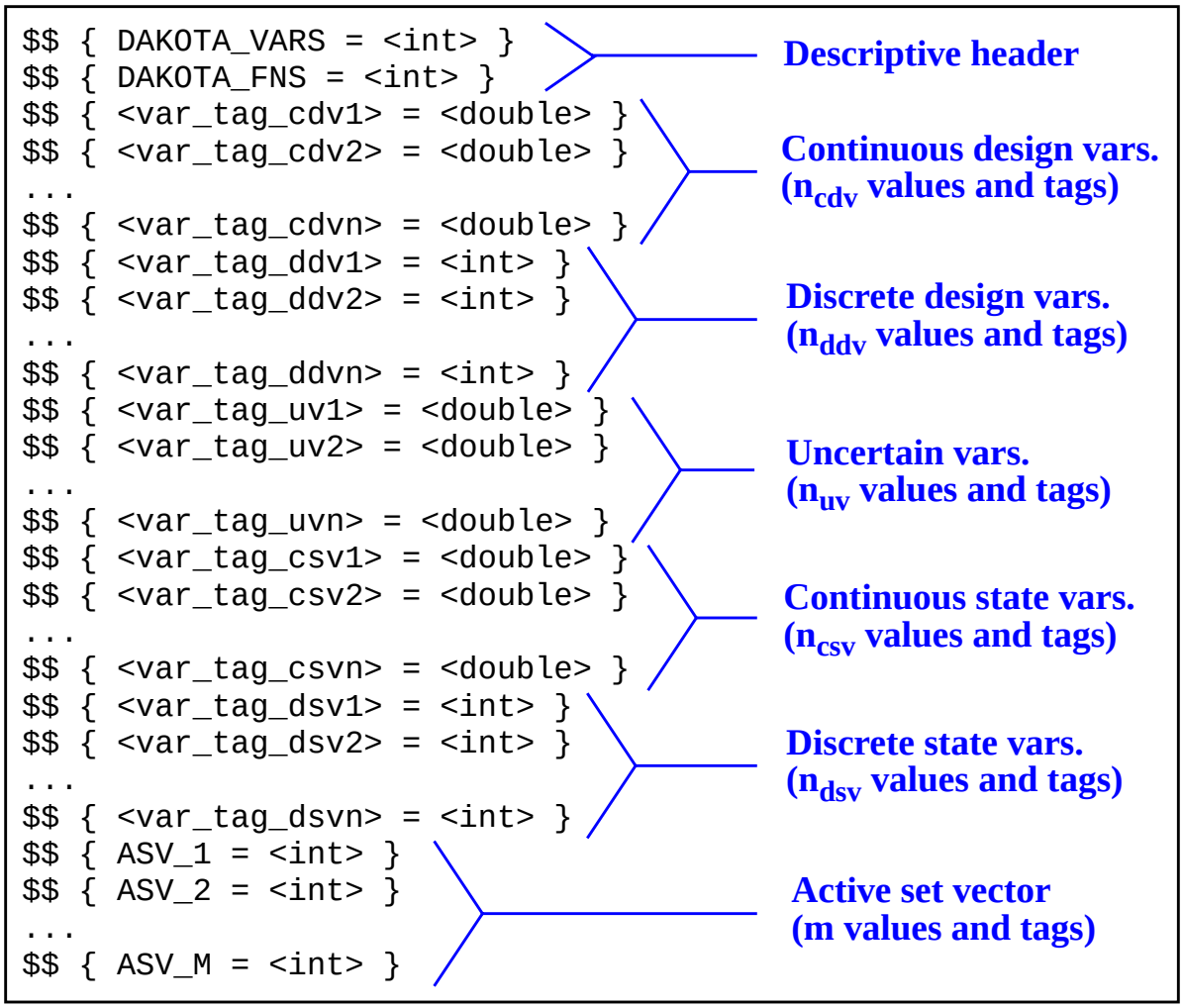


Figure 21 Parameters file data format, APREPRO option

When a parameters file in APREPRO format is included within a template file (using an include directive), the APREPRO utility recognizes these constructs as variable definitions which can then be used to populate targets throughout the template file.

Results file format

After completion of the interfacing processes, DAKOTA expects to read a file containing response data for the current set of parameters and corresponding to the set of function requests. This data must be in the following format:

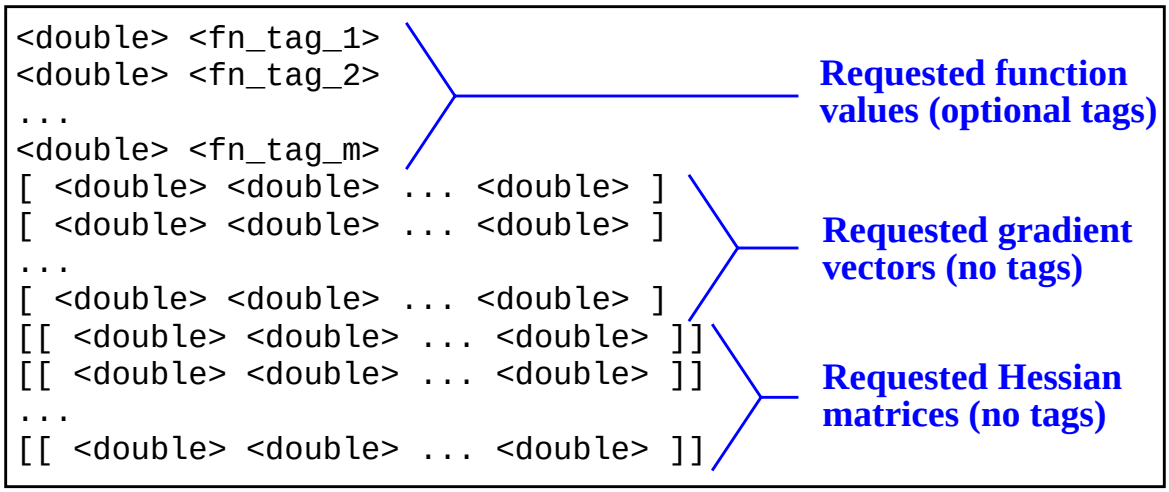


Figure 22 Results file data format

The first block of data is the function values that have been requested, followed by a block of requested gradient data, followed by a block of requested Hessian data. Function data have no bracket delimiters and 1 character tag per function can be *optionally* supplied. These tags are not used by DAKOTA and are only included as an optional field for consistency with the parameters file format and for backwards compatibility. The tags are rendered optional through DAKOTA's use of regular expression pattern matching to detect whether an upcoming field is numerical data or a tag. If character tags are used, then they must be separated from data by either white space or new line characters and there must not be any white space within a character tag (e.g., use "variable_1," not "variable 1").

Function gradient vectors are delimited with single brackets [...n_{grad}-vector of doubles...]. Tags are not used and must not be present. White space separating the brackets from the data is optional.

Function Hessian matrices are delimited with double brackets [[...$n_{grad} \times n_{grad}$ matrix of doubles...]]. Tags are not used and must not be present. White space separating the brackets from the data is optional, although white space must not appear between the double brackets.

DAKOTA will read the data in three passes, getting the set of requested function values first, followed by the requested set of gradients, followed by the requested set of Hessians. If the amount of data in the file does not match the function request vector, DAKOTA will abort with a response recovery format error message.

An important question for proper management of both gradient and Hessian data is: if several different types of variables are used, ***for which variables are function derivatives needed?*** That is, how is n_{grad} determined? Derivatives are never needed with respect to any discrete variables (since these derivatives do not exist) and the types of continuous variables for which derivatives are needed depend on the type of study being performed. For optimization and least squares problems, function derivatives are only needed with respect to the *continuous design variables* (n_{grad}=n_{cdv}) since this is the information used by the optimizer in computing a search

direction. Similarly, for nondeterministic analysis methods which use gradient and/or Hessian information, function derivatives are only needed with respect to the *uncertain variables* ($n_{grad}=n_{uv}$). And lastly, parameter study methods which are cataloguing gradient and/or Hessian information do not draw a distinction among continuous variables; therefore, function derivatives must be supplied with respect to *all continuous variables* that are specified ($n_{grad}=n_{cdv}+n_{uv}+n_{csv}$). This is generally not as complicated as it sounds, since it is common for optimization and least squares problems to only specify design variables and for nondeterministic analysis problems to only specify uncertain variables. DAKOTA allows for the specification of additional types of variables in these cases and DAKOTA will map these additional variables through the interface, but since they will not be used in the internal computations of the iterator, the derivatives of the function set with respect to the additional variables are not needed.

Active set vector control

A future capability will be the option to turn the ASV control `on` or `off` (currently, `dakota.input.spec` has a placeholder for this capability in the responses keyword section). ASV control set to `on` is the default operation as described previously, whereas ASV control set to `off` will cause Dakota to always request a "full" data set (the full function, gradient, and Hessian data that is available in the problem as specified in the responses specification) on each function evaluation. This latter case will allow the user to simplify the supplied interface by removing the need to check the content of the active set vector on each evaluation. Of course, this will be most appropriate for those cases in which only a relatively small penalty in efficiency occurs when returning more data than may be needed on a particular function evaluation. See Active Set Vector Usage on page 143 in the Responses section of the Commands chapter for a more detailed description.

Examples

Shown are several examples of parameters files and their corresponding results files.

A typical input file for 2 variables (`n` = 2) and 3 functions (`m` = 3) is as follows:

```
2 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
 1 ASV_1
 1 ASV_2
 1 ASV_3
```

The number of design variables (`n`) and the string "variables" are followed by the number of functions (`m`) and the string "functions", the values of the design variables and their tags, and the active set vector (ASV) and its tags. The descriptive tags for the variables are always present and they are either the descriptors specified in the user's dakota input file or are default descriptors if none were provided. The length of the active set vector is equal to the number of functions (`m`). In the case of an optimization data set with an objective function and two nonlinear constraints (three response functions total), the first ASV value is associated with the objective function and

the remaining two are associated with the constraints (in whatever consistent order has been defined by the user).

For the APREPRO format option, the same set of data appears as follows:

```
$$ { DAKOTA_VARS     = 2 }
$$ { DAKOTA_FNS      = 3 }
$$ { cdv_1           =  1.5000000000e+00 }
$$ { cdv_2           =  1.5000000000e+00 }
$$ { ASV_1           =                 1 }
$$ { ASV_2           =                 1 }
$$ { ASV_3           =                 1 }
```

where the numerical values are associated with their tags within `$${ tag = value }` constructs.

The user-supplied application interface, comprised of a simulator program and - optionally - filter programs, is responsible for reading the parameters file and writing the results file containing the response data requested in the ASV. Since the ASV contains all ones in this case, the response file corresponding to the above input file would contain values for the three functions:

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
```

Since function tags are optional, the following would be equally acceptable:

```
1.2500000000e-01
1.5000000000e+00
1.7500000000e+00
```

For the same parameters with different ASV components,

```
 2 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
 3 ASV_1
 3 ASV_2
 3 ASV_3
```

the following response data is required:

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
[ 5.0000000000e-01 5.0000000000e-01 ]
[ 3.0000000000e+00 -5.0000000000e-01 ]
[ 0.0000000000e+00 3.0000000000e+00 ]
```

Here, we need not only the function values, but also each of their gradients. Modifying the ASV components again gives the following parameters file,

```
 2 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
 2 ASV_1
 0 ASV_2
```

```
  2 ASV_3
```

for which the following results file is needed:

```
[ 5.0000000000e-01 5.0000000000e-01 ]
[ 0.0000000000e+00 3.0000000000e+00 ]
```

Here, we needed gradients for functions `f` and `c2`, but not for `c1` presumably since the constraint is inactive.

A full Newton optimizer might well make the following request:

```
 2 variables 1 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
 7 ASV_1
```

for which the following results file (containing the objective function, its gradient vector, and its Hessian matrix) is needed:

```
1.2500000000e-01 f
[ 5.0000000000e-01 5.0000000000e-01 ]
[[ 3.0000000000e+00 0.0000000000e+00 0.0000000000e+00
     3.0000000000e+00 ]]
```

Lastly, a more advanced example might have multiple types of variables present:

```
 11 variables 3 functions
1.5000000000e+00 cdv_1
1.5000000000e+00 cdv_2
 2 ddv_1
 2 ddv_2
 2 ddv_3
3.5000000000e+00 csv_1
3.5000000000e+00 csv_2
3.5000000000e+00 csv_3
3.5000000000e+00 csv_4
 4 dsv_1
 4 dsv_2
 3 ASV_1
 3 ASV_2
 3 ASV_3
```

In this case, the required length of the gradient vectors depends upon the type of study being performed. In an optimization problem, gradients are only needed with respect to the continuous design variables, in which case the following response data would be appropriate (n_{grad}=2):

```
1.2500000000e-01 f
1.5000000000e+00 c1
1.7500000000e+00 c2
[ 5.0000000000e-01 5.0000000000e-01 ]
[ 3.0000000000e+00 -5.0000000000e-01 ]
[ 0.0000000000e+00 3.0000000000e+00 ]
```

In a parameter study, however, no distinction is drawn between different types of continuous variables and gradients would be needed with respect to all continuous variables (n_{grad}=6), e.g.:

```
1.2500000000e-01 f
```

```
1.5000000000e+00 c1
1.7500000000e+00 c2
[ 5.0000000000e-01 5.0000000000e-01 6.2500000000e+01
     6.2500000000e+01 6.2500000000e+01 6.2500000000e+01 ]
[ 3.0000000000e+00 -5.0000000000e-01 0.0000000000e+00
     0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 ]
[ 0.0000000000e+00 3.0000000000e+00 0.0000000000e+00
     0.0000000000e+00 0.0000000000e+00 0.0000000000e+00 ]
```

Failure capturing

DAKOTA provides the capability to manage failures in simulation codes within both its system call and direct application interfaces. Failure capturing consists of three operations: failure detection, failure communication, and failure recovery.

Failure detection

Since the symptoms of a simulation failure are highly code-dependent, it is the user's responsibility to detect failures within their `analysis_driver` or `output_filter`. One popular example of simulation monitoring is to rely on a simulation's internal detection of errors. In this case, the Unix `grep` utility can be used within a user's script to detect strings in output files which indicate analysis failure. For example, the following script excerpt

```
grep ERROR analysis.out > /dev/null
if ( $status == 0 )
  echo "FAIL" > results.out
endif
```

will pass the `if` test and communicate simulation failure to DAKOTA if the `grep` command finds the string `ERROR` anywhere in the `analysis.out` file.

If the simulation code is not providing error diagnostic information, then failure detection may require monitoring of simulation results for sanity (e.g., is the mesh distorting excessively?) or potentially monitoring for continued process existence to detect a simulation segmentation fault or core dump. While this can get complicated, the flexibility of DAKOTA's interfaces allows for a wide variety of monitoring approaches.

Failure communication

Once a failure is detected, it must be communicated so that DAKOTA can attempt to recover from the failure. The form of this communication depends on the type of application interface in use.

System call application interfaces

In the system call application interface case, a detected simulation failure is communicated to DAKOTA through the results file returned by the user's `analysis_driver` (1-piece interface) or `output_filter` (3-piece interface). Instead of returning the standard results file data, the string "`FAIL`" or "`fail`" should appear at the beginning of the results file. Any data appearing after the fail string will not be read.

Direct application interfaces

In the direct application interface case, a detected simulation failure is communicated to DAKOTA through the return code provided by the user's `analysis_driver` (for either the 1-piece or the 3-piece interface). Recall that the prototype for the direct interface is `int function_name(const DakotaVariables& vars, const DakotaIntArray& asv, DakotaResponse& response)`. The `int` returned is the failure code: 0 (false) if no failure occurs and 1 (true) if a failure occurs.

Failure recovery

Once the analysis failure has been communicated, DAKOTA will attempt to recover from the failure using one of the following mechanisms, as governed by the user's input specification. Additional details on these specifications are provided in Interface Commands on page 127.

Abort

If the `abort` option is specified, then DAKOTA will terminate upon detecting a failure. Note that if the problem causing the failure can be corrected, DAKOTA's restart capability (see Restart Management on page 125) can be used to continue the study.

Retry

If the `retry` option is specified, then DAKOTA will reinvoke the failed simulation up to the specified number of retries. If the simulation continues to fail on each of these retries, DAKOTA will terminate. The retry option is appropriate for those cases in which simulation failures may be resulting from transient computing environment issues, such as disk space.

Recover

If the `recover` option is specified, then DAKOTA will not attempt the failed simulation again. Rather, it will return a "dummy" set of function values as the results of the function evaluation. The dummy function values to be returned are specified by the user. Any gradient or Hessian data requested in the active set vector will be zero. This option is appropriate for those cases in which a failed simulation may indicate a region of the design space to be avoided and the dummy values can be used to return a large objective function or a constraint violation which will discourage an optimizer from further investigating the region.

Continuation

If the `continuation` option is specified, then DAKOTA will attempt to step towards the failing “target” simulation from a nearby “source” simulation through the use of a continuation algorithm. This option is appropriate for those cases in which a failed simulation may be caused by an inadequate initial guess. If the “distance” between the source and target can be divided into smaller steps in which information from one step provides an adequate initial guess for the next step, then the continuation method can step towards the target in increments sufficiently small to allow for convergence of the simulations.

When the failure occurs, the interval between the last successful evaluation (the source point) and the current target point is halved and the evaluation is retried. This halving is repeated until a successful evaluation occurs. The algorithm then marches towards the target point using the last interval as a step size. If a failure occurs while marching forward, the interval will be halved again. Each invocation of the continuation algorithm is allowed a total of ten failures (ten halvings result in up to 1024 evaluations from source to target) prior to aborting the DAKOTA process.

While DAKOTA manages the interval halving and function evaluation invocations, the user is responsible for managing the initial guess for the simulation program. For example, in GOMA ([Schunk, P.R., Sackinger, P.A., Rao, R.R., Chen, K.S., Cairncross, R.A., 1995]), the user specifies the files to be used for reading initial guess data and writing solution data. When using the last successful evaluation in the continuation algorithm, the translation of initial guess data can be accomplished by simply copying the solution data file leftover from the last evaluation to the initial guess file for the current evaluation (and in fact this is useful for all evaluations, not just continuation). However, techniques are under development for use of the *closest*, previously successful, function evaluation (rather than the *last* successful evaluation) as the source point in the continuation algorithm. This will be especially important for nonlocal methods (e.g., genetic algorithms) in which the last successful evaluation may not necessarily be in the vicinity of the current evaluation. This approach will require the user to save and manipulate previous solutions (likely tagged with evaluation number) so that the results from a particular simulation (specified by DAKOTA after internal identification of the closest point) can be used as the current simulation’s initial guess.

The Approximation Interface

The **ApproximationInterface** branch (see Figure 18) implements a variety of approximation techniques which can be used as surrogates in place of actual simulations. The **ANN**, **RSM**, and **MARS** approximation interfaces implement artificial neural networks, response surface methods, and multivariate adaptive regression splines, respectively. In addition, an **MPA** approximation interface is planned for implementing multipoint approximations. These approximations can be used on their own for direct interfacing with any iterator or as part of a sequential approximation strategy (see Sequential Approximate Optimization on page 74).

The primary goal in surrogate-based optimization is the reduction of computational expense through the minimization of the number of function evaluations that need to be performed with the actual expensive model.

All of the approximation interfaces define methods for building an initial approximation (the `build_approximation` virtual function), updating the approximation with new data points (the `update_approximation` virtual function), modifying the form or extent of the approximation (the `modify_approximation` virtual function), and performing a function evaluation using the approximation (the `map` virtual function).

Building an approximation

Building an initial approximation consists of selecting a set of trial points, performing the trial function evaluations on the actual model, and then using the results of the trial function evaluations to solve for the coefficients (e.g., polynomial coefficients, neural network weights) of the approximation. If there are multiple functions in the response set (e.g., an objective function plus one or more constraints), then a separate approximation is built for each function, although each approximation uses the response data from the same trial points. Currently, only 0^{th}-order information (function values) from the actual model is used in building the approximation, although extensions to using higher-order information (function gradients and Hessians) are possible. In DAKOTA, the set of trial points is determined via the DDACE package ([Tong, C.H., and Meza, J.C., 1997]) for design and analysis of computer experiments. Solution for the approximation coefficients is performed using either LU factorization or singular value decomposition.

Updating an approximation

An approximation can be updated whenever new information is available from the actual model. In sequential approximate optimization, for example, the best point found in an approximate optimization cycle is evaluated with the actual model. This new information is first used to assess performance and convergence of the process. If improvement is observed and the convergence criteria have not been satisfied, then the new function evaluation information is used to update the approximation for the next approximate cycle. This will typically involve another factorization or decomposition to solve for new approximation coefficients.

Modifying an approximation

It is often desirable to modify the extent of an approximation based on its performance. For example, if the approximation is performing poorly (as measured by the evaluation of the best point found in an approximate optimization cycle with the actual model), then it is desirable to restrict the extent (i.e., the bounds) of the approximation. Conversely, if the approximation is performing well, then it may be desirable to increase the extent of the approximation so that

larger changes can occur on each cycle. DAKOTA is implementing trust region concepts to manage the extent of approximations.

Performing function evaluations

Each of the approximation interfaces, like the application interfaces, must implement the virtual `map` function in order to provide a mechanism for parameter to response mapping. This is the function invoked when an iterator requests a function evaluation. Since the function evaluation mechanisms for application and approximation interfaces are implemented within a single virtual function, the particular form of the interface can be hidden from the iterator and this complexity can be encapsulated.

In the case of an approximation interface, a parameter to response mapping involves an inexpensive evaluation of the approximation for a particular parameter set. All of the approximations can return 0^{th}-order information (approximate function values) and some approximations can directly return 1^{st}-order information (approximate function gradients) in those cases where the approximate form is easily differentiated (e.g., a quadratic polynomial approximation). Availability of analytic gradients can improve the accuracy and efficiency of performing a gradient-based optimization on the approximation.

The RSM Approximation Interface

The RSM Approximation Interface uses a response surface method which assumes a quadratic polynomial of the form:

$$c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_i x_j \qquad (7)$$

Following evaluation of the DDACE sample points with the actual model, the RSM approximation coefficients (c_0, c_i, c_{ij}) are computed with an LU factorization.

This capability is new and evolving. Additional details will be provided in future documentation releases.

The MARS Approximation Interface

The MARS Approximation Interface uses multivariate adaptive regression splines from the MARS3.5 package ([Friedman, J. H., 1991]) developed at Stanford University. An object-oriented interface to the Fortran library is provided by the DDACE package ([Tong, C.H., and Meza, J.C., 1997]).

This capability is new and evolving. Additional details will be provided in future documentation releases.

The ANN Approximation Interface

The ANN Approximation Interface uses a layered perceptron artificial neural network based on the direct training approach of Zimmerman ([Zimmerman, D.C., 1996]). Following evaluation of the DDACE sample points with the actual model, the ANN weights are computed with an SVD decomposition.

This capability is new and evolving. Additional details will be provided in future documentation releases.

Exploiting Parallelism

parallelism on page 103

Parallelism Introduction

The opportunities for exploiting parallelism in optimization can be categorized into four main areas:

1. *Algorithmic coarse-grained parallelism*: This parallelism involves the exploitation of multiple independent function evaluations. Examples of optimization algorithms containing coarse-grained parallelism include:
   a.) *Gradient-based algorithms:* finite difference gradient evaluations, speculative optimization, parallel line search, multiple-secant BFGS.
   b.) *Nongradient-based algorithms:* genetic algorithms (GA's), coordinate pattern search (CPS), parallel direct search (PDS), Monte Carlo.
   c.) *Approximate methods:* design and analysis of computer experiments (DACE) evaluations for building response surfaces and training neural networks.
   d.) *Multi-method strategies:* optimization under uncertainty, branch and bound, multi-start local search, island-model GA's, GA's with periodic local search.

2. *Algorithmic fine-grained parallelism:* This involves computing the basic computational steps of an optimization algorithm (i.e., the internal linear algebra) in parallel. This is primarily of interest in large-scale optimization problems and simultaneous analysis and design (SAND).

3. *Function evaluation coarse-grained parallelism:* This involves simultaneous computation of separable (i.e., uncoupled) parts of a single function evaluation, where a function evaluation may contain multiple response functions requiring multiple simulations. Examples include separate simulations for multiple objectives and constraint functions, multiple disciplinary analyses for MDO, etc.

4. *Function evaluation fine-grained parallelism:* This involves parallelization of the solution steps within a single analysis code. Examples of Sandia-developed MP analysis codes include PRONTO3D, COYOTE, MPSalsa, ALEGRA, PCTH, SIERRA, etc.

In both the algorithmic and function evaluation cases, coarse-grained parallelization requires very little inter-processor communication and is therefore essentially "free," meaning that there is little loss in parallel efficiency due to communication as the number of processors increases (assuming that there are enough separable computations to utilize the additional processors). Fine-grained parallelism, on the other hand, involves much more communication among processors and care must be taken to avoid the case of inefficient machine utilization in which the communication demands among processors outstrip the amount of actual computational work to be performed.

Single-level approaches which exploit either algorithmic coarse-grained parallelism or function evaluation fine-grained parallelism have been investigated in previous work ([Eldred, M.S., Hart,

W.E., Bohnhoff, W.J., Romero, V.J., Hutchinson, S.A., and Salinger, A.G., 1996]). It has been shown that optimization approaches which utilize single-level parallelism can have clear performance barriers. Parallel optimization of single-processor simulations is limited by the number of independent evaluations per cycle, and sequential optimization of parallel analyses is limited by the practical limit on processors that can be used for a single parallel simulation before inter-process communication dominates actual computational work. These observations point clearly to the need for multilevel parallelism, in which parallel optimization strategies coordinate multiple simultaneous simulations of multiprocessor codes.

The question arises, then, if multiple types of parallelism can be exploited, how should the amount of parallelism at each level be selected so as to maximize the parallel efficiency of the study? This question is answered in [Eldred, M.S., and Hart, W.E., 1998] in which it is shown that maximum parallel efficiency is achieved in multilevel parallelism when the minimum number of processors is used for the fine-grained parallelism of a given parallel analysis (with the rare exception of a parallel analysis with superlinear speedup). This gives preference to the coarse-grained parallelism in multilevel parallel studies. However, maximum efficiency and minimum turn-around time are not equivalent, and in practice, it is common to sacrifice efficiency for speed and increase the number of processors used for a given parallel analysis beyond the minimum required. For example, if an algorithm has 10 independent function evaluations per cycle and each of these function evaluations needs a minimum of 50 processors to perform the simulation, then high parallel efficiency can be achieved by dividing a total of 501 processors into ten 50-processor slave servers plus a master processor. This would be preferable to five 100-processor slave servers and far preferable to one 500-processor slave server. However, increasing to a total of 1001 processors and selecting 10 100-processor slave servers, while not having as high a parallel efficiency, might be desirable in practice in order to minimize turn-around time.

The following discussions describe how to manage algorithmic coarse-grained parallelism and function evaluation fine-grained parallelism within the DAKOTA framework. The remaining types (algorithmic fine-grained and function evaluation coarse-grained parallelism) are not currently supported, although [Eldred, M.S., and Schimel, B.D., 1999] describes recent progress in these directions. The software components which enable parallelism are discussed first, followed by descriptions of approaches for utilizing these components in implementing parallelism within a variety of scenarios. Finally, input specification and execution details are provided for running parallel DAKOTA studies.

Enabling Software Components

This section describes software components which enable parallelism in a variety of forms. Direct function and system call interfacing capabilities have the flexibility to initiate function evaluations either synchronously or asynchronously. Synchronous evaluations proceed one at a time with the evaluation running to completion before control is returned to DAKOTA. Asynchronous evaluations are initiated such that control is returned to DAKOTA immediately,

prior to evaluation completion, thereby allowing the initiation of multiple concurrent evaluations. The synchronization capabilities can be used by themselves to provide a simple parallelism which relies on external means to assign jobs to processors (see Single-processor DAKOTA implementation on page 105), or they can be combined with DAKOTA's master-slave algorithm to provide a sophisticated self-contained parallelism (see Multiprocessor DAKOTA implementation on page 106).

Direct function synchronization

The direct function capability, described in detail in The Direct Function Application Interface on page 80, is used to invoke simulation codes which are linked directly into the DAKOTA executable or to invoke internal test functions for algorithm performance testing. This capability may be used synchronously or asynchronously:

Synchronous

Synchronous operation of the direct function application interface involves a standard procedure call to a simulation linked within the code. Control does not return to the calling code until the simulation is completed and the response object has been populated.

Asynchronous

Asynchronous operation involves the use of multithreading (e.g., POSIX threads) to accomplish multiple simultaneous simulations. When spawning a thread, control returns to the calling code after the simulation is initiated. In this way, multiple threads can be created simultaneously. An array of responses corresponding to the multiple threads of execution is recovered in a synchronize operation.

System call synchronization

The system call approach, described in detail in The System Call Application Interface on page 81, invokes a simulation code or simulation driver by using the `system` function from the C standard library to create a new process. This capability may be used synchronously or asynchronously:

Synchronous

Synchronous operation of the system call application interface involves spawning the system call in the foreground. Control does not return to the calling code until the simulation is completed and the response file has been written. In this case, the possibility of a race condition (see below) does not exist and any errors during response recovery will cause an immediate abort of the DAKOTA process.

Asynchronous

Asynchronous operation involves spawning the system call in the background, continuing with other tasks (e.g., other simulation system calls), periodically checking for process completion, and finally retrieving the results. An array of responses corresponding to the multiple system calls is recovered in a synchronize operation.

In this synchronize operation, completion of a function evaluation is detected by testing for the existence of the evaluation's results file using the `stat` utility (see [Kernighan, B.W., and Ritchie, D.M., 1988]). Care must be taken when using this facility since it is prone to the race condition in which the results file passes the existence test but the recording of the function evaluation results in the file is incomplete. In this case, the read operation performed by DAKOTA will result in an error due to this incomplete data set. In order to address this problem, DAKOTA contains exception handling which allows for a fixed number of response read failures per asynchronous system call evaluation. The number of allowed failures must have a limit, so that an actual response format error (unrelated to the race condition) will eventually abort the system. Therefore, to reduce the possibility of exceeding the limit on allowable read failures, ***the user's interface should minimize the amount of time an incomplete results file exists in the directory where its status is being tested.*** This can be accomplished through two approaches: (1) delay the creation of the results file until the simulation computations are complete and all of the response data is ready to be written to the results file, or (2) perform the simulation computations in a subdirectory, and as a last step, move the completed results file into the main working directory where its existence is being queried.

If concurrent simulations are executing in a shared disk space, then care must be taken to maintain independence of the concurrent analyses. In particular, the parameters and results files for a simulation, as well as any other files used by the simulation, must be protected from other files of the same name used by other concurrent simulations. With respect to the parameters and results files, these files may be made unique through the use of file tagging or Unix temporary files (see Additional Features on page 82). However, if additional simulation files must be protected, then it will usually be necessary to create a working subdirectory for each concurrent simulation. For example, if the only files used by a simulator are the files from which it reads parameters and to which it writes results (e.g., the simple test problems in Example Problems on page 328), then it is sufficient to use either the `file_tag` option (`params.in.1`, `results.out.1`, etc.) or the default Unix temporary file option (`/var/tmp/aaa0b2Mfv`, etc.) to maintain independence between concurrent simulations. If, however, a simulator needs to use additional files for input, run diagnostics, and results databases (e.g., `model.i`, `model.o`, `model.g`, `model.e`, etc., for many SEACAS codes), then one could extract DAKOTA's number designators and use them to tag all the other files (assuming the simulator can handle modified filenames), or preferably, create a tagged working directory in which the simulator can execute in default mode. An example of this preferred approach is given in Figure 9 in the Tutorial on page 19.

Master-slave algorithm

DAKOTA contains a master-slave algorithm which self-schedules function evaluations in a "single program-multiple data" (SPMD) parallel programming model. It uses MPI message-passing ([Gropp, W., Lusk, E., and Skjellum, A., 1994], [Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J., 1996]) to communicate data between processors. The self-scheduling design (also known as a task pool design) provides a simple load balancing which is particularly useful in the case of heterogeneous processor speeds or varying simulation durations. In the first pass, the self-scheduling algorithm assigns each slave server a job. In the second pass, the master schedules the remaining jobs on slave servers as they complete their previous jobs; the first server to return its results gets the next job. The SPMD designation simply denotes that the same DAKOTA executable is loaded on all processors. This differs from the MPMD model ("multiple program-multiple data") which would have the DAKOTA executable on the master processor communicating directly with simulator executables on slave processors. The MPMD model has some advantages, but it is not currently allowable by the executable loading software (i.e., `yod`) on Sandia's MP machines.

Developer's notes: Implementing the master-slave model within a single executable (SPMD model) entails a division of iterator code (master) from function evaluation code (slave). This is accomplished within DAKOTA at the strategy layer. In the strategy constructor, the master processor instantiates the required iterators and models whereas the slave processors instantiate only the required models. When the strategy is executed, the master executes the current iterator and sends analysis requests to the slaves which run server code bound to the current model. When the master completes iteration on the current model, it sends a termination message to the slaves which then exit the current model. If additional work remains within the strategy, then the process repeats for the next iterator and model. Additional features include: (1) the use of buffer packing which allows for send/receive of a heterogeneous set of data within a single message and thereby minimizes message traffic, and (2) use of a **ParallelLibrary** class hierarchy which encapsulates the specific syntax of message passing operations for particular message passing libraries.

Single-level parallelism

DAKOTA uses MPI communicators to identify groups of processors. In the single-level parallel case employing many single-processor slave servers, the global MPI communicator (`MPI_COMM_WORLD`) can directly provide the context needed for master-slave communication since processor rank within `MPI_COMM_WORLD` is sufficient for message source and destination information. The other single-level parallel case of employing one multi-processor slave server is treated identically to the multilevel parallel case described below.

Multilevel parallelism

For multilevel parallelism, `MPI_COMM_WORLD` can be partitioned into new intra-communicators which delineate the set of processors to be used for each multiprocessor analysis. Since these intra-communicators can be passed into a simulation for use as the simulation's computational context, the use of communicators enables the analysis routines to be provided as a generic library utility that can be run on an arbitrary set of processors (which was one of the goals of the MPI standard). Within DAKOTA, new intra-communicators are created with the `MPI_Comm_split` routine. In order for the master to send messages to the new intra-communicators, inter-communicators are created with calls to `MPI_Intercomm_create`.

Once the new communicators are created, the single-level and multilevel algorithms for scheduling jobs from the master are virtually identical (in fact, the single-level case could be handled as a special case of the multilevel case, but the DAKOTA design opted to maintain separate logic and avoid the overhead of additional communicator creations for the single-level case). In addition, communicator partitions can be reallocated multiple times. This enables dynamic repartitioning of `MPI_COMM_WORLD` for each simulation interface within a strategy that manages multiple models (e.g., four 256 processor servers could be used for a coarse model followed by two 512 processor servers for subsequent iteration on a fine model). This is conveniently managed by allocating and deallocating particular communicator partitioning schemes within the iterator/model loops of the strategy layer.

Pending Extensions

Recent work has focused on the development of concurrent-iterator strategies and concurrent-analysis function evaluations (refer to [Eldred, M.S., and Schimel, B.D., 1999]) for exploiting additional coarse-grained parallelism within optimization studies. These extensions result in a total of three nested tiers of master-slave control and four levels of parallelism which can minimize efficiency losses and achieve near linear scaling on massively parallel computers. These capabilities will be available in the DAKOTA V1.2 release and will allow the convenient selection and combination of each type of parallelism a particular application supports:

- Concurrent iterators within a strategy
- Concurrent function evaluations within an iterator
- Concurrent analyses within a function evaluation
- Multiprocessor analyses

Implementation of Parallelism

This section describes how the software components which enable parallelism can be configured to perform particular parallel studies. An essential feature for enabling a variety of parallel processing scenarios is the independence of the Master-slave algorithm on page 103 from the interfacing software described in Direct function synchronization on page 101 and System call synchronization on page 101. Since they are independent, the master-slave code can utilize any of the available interfacing capabilities, or alternatively, any of the available interfacing capabilities can be employed with or without the master-slave approach.

The approaches to exploiting parallelism which this flexibility allows can be categorized into two main areas: those in which DAKOTA runs on a single processor and relies on external means to distribute simulations to remote processors (the master-slave approach is *not* used), and those in which DAKOTA runs in parallel coordinating simulations within its allocation of processors (the master-slave approach *is* used).

Single-processor DAKOTA implementation

The asynchronous mappings described in Direct function synchronization on page 101 and System call synchronization on page 101 can be used to accomplish coarse-grained parallelism even when the DAKOTA process is running on a single processor. In this case, the master-slave algorithm is not used and jobs are not assigned with MPI message-passing. Therefore, some additional mechanism external to DAKOTA will usually be desired to distribute the asynchronous jobs among processors, since multitasking on a single processor is generally slower than running the jobs sequentially. For the asynchronous system call case, network load leveling software (e.g., load leveler, load sharing facility, or other native scheduling software) or compute server job queues can provide this mechanism, and in the asynchronous direct function case, thread schedulers can be used (e.g., to select nodes within an SMP architecture).

To accomplish multilevel parallelism in this context, one could use DAKOTA's asynchronous system call interface to submit multiple multiprocessor jobs to the queues of a parallel compute server. Unfortunately, loading the queues with multiple jobs is generally forbidden in the usage rules of Sandia's MP machines. Moreover, each set of concurrent jobs will suffer a delay while it percolates through the queue, such that an optimization performing evaluations in this way suffers repeated queue delays on each cycle (as opposed to a single queue delay in other approaches). Nevertheless, if specialized queues which allow multiple jobs per user and which minimize repeated delays can be created and balanced with competing concerns, then this approach can be a viable avenue to multilevel parallelism.

An alternative approach is to allocate a large number of compute processors to a single script which runs on a service node and manages concurrent multiprocessor jobs on partitions of the total allocation. This is in fact mimicking the communicator partitioning capabilities of MPI within sophisticated scripting. While this has the advantages of simplifying the automation of pre- and post-processing (since service nodes run full Unix) and minimizing analysis code modifications (since the analysis does not have to be modified to a callable subroutine), it has the disadvantages that (1) this is highly specific to the job submission software of a particular parallel machine and is therefore not particularly flexible or extensible, and (2) DAKOTA is disconnected from its function evaluations. This disconnection is due to the fact that DAKOTA and the server script are launched separately, and information normally passed to the simulations by DAKOTA during simulation invocation (e.g., where to obtain the parameters and where to write the results) must be mimicked by the server script. DAKOTA's only communication with the simulations in this case comes through the creation of parameters files and the capture of completed results files. While this procedure has been successfully demonstrated for a single multiprocessor simulation, concurrent multiprocessor simulations will have the additional complication that the server script must correctly track the evaluation numbers (which are not a simple increasing sequence is the presence of duplicated analyses) in order to associate the proper tagged files from DAKOTA with the analyses it launches.

The final option for multilevel parallelism is to use the multiprocessor DAKOTA implementation (described in the following section) and manage multiprocessor function evaluations internally. While elegant and general-purpose, it also has disadvantages in required modification to analysis

codes. Each of these three options is currently under investigation, and it is expected that future releases of the software documentation will be able to recommend the most fruitful of these approaches.

Multiprocessor DAKOTA implementation

When executing DAKOTA in multiprocessor mode using the Master-slave algorithm on page 103, the synchronous and asynchronous operations of the direct function and system call simulation interfacing classes are issues that are local to a processor. Layered on top of these local interfacing capabilities is the software which manages message passing for assignment of work among processors. This design allows flexibility in handling local evaluation mechanisms independently from the particular form of the global message passing model. For example, within the global context of a master-slave approach in which the master is *asynchronously* assigning jobs and retrieving results using message passing with slave servers, the slave servers locally execute their simulations using the *synchronous* direct function or system call protocols. This is due to the fact that, since the master-slave algorithm is managing the parallelism and scheduling one job at a time to a server, there is nothing to be gained in performing the job asynchronously on the server. However, if new approaches or architectures become available which can exploit additional parallelism at the slave server level (e.g., message-passing across multiple SMP's with multiple asynchronous jobs on each SMP), then the asynchronous direct function and system call capabilities could be employed to realize this additional parallelism.

In the single-level parallel case of single-processor analyses, either the system call or direct function interfacing approaches can be used. The system call case is particularly popular on clusters of workstations since the analysis can be used in unmodified form and the user can employ a simple driver script to coordinate any combination of pre- and post-processing tools associated with an analysis. Applications can be configured quickly and easily in this way.

For multiprocessor analyses, the system call interfacing approach cannot be used since it is not possible to share an MPI communicator (which provides the computational context for the multiprocessor analysis) between processes spawned with system calls on different processors. Therefore, the direct function interfacing approach must be used whenever employing multiprocessor analyses within multiprocessor DAKOTA (in this case, an MPI communicator can be passed in through the procedure call for all processors within a slave server - see Multilevel parallelism on page 103 for additional MPI details). The main ramification of the restriction to the direct function interface is the requirement to modify the analysis into a callable subroutine and link it into the executable (see The Direct Function Application Interface on page 80). However,it may be feasible to remove this restriction in the future through use of MPMD ("Multiple Program, Multiple Data") executable loading or dynamic process creation with the emerging MPI-2 standard (see [Eldred, M.S., and Hart, W.E., 1998] for additional details).

Specifying Parallelism

In specifying parallelism with DAKOTA, the “model” encompasses the parallelism that is supported in the problem (in particular, the interface specification specifies the available parallelism). Then, depending on the “iterator” selected, the available parallelism (i.e., multiple processors, asynchronous interfaces) will be automatically exploited in particular ways. This design is known as ***implicit parallelism***, in that the use of parallelism by an iterator is implicit: the methods recognize the available parallelism and exploit it without need for specification of special parallelized methods.

The Model

Specifying parallelism within a model can involve the use of the `asynchronous`, `evaluation_servers`, and `processors_per_evaluation` keywords described in Interface Commands on page 127.

When using DAKOTA on a single-processor and relying on external means to allocate jobs to processors (see Single-processor DAKOTA implementation on page 105), the `asynchronous` interface specification is all that is required to specify the availability of asynchronous system calls or asynchronous direct invocations within a model.

When executing DAKOTA across multiple processors and managing job allocation internally (see Multiprocessor DAKOTA implementation on page 106), DAKOTA automatically detects the presence of multiple processors and will, by default, *asynchronously* schedule jobs among slave processors while executing the jobs on the slave processors using *synchronous* invocations. If the function evaluations are to be performed on multiple processors (multilevel parallelism), then `evaluation_servers` or `processors_per_evaluation` must be specified to define how the total processor allocation will be partitioned into function evaluation servers for a particular simulation interface.

Note: asynchronous execution on the slave processors may be supported in the future for SMP clusters and will be triggered by the `asynchronous` interface specification. However, using this specification in multiprocessor mode is not supported in the current release.

The Iterator

As mentioned previously, iterators automatically detect the parallelism available in a model and exploit it as appropriate within the iteration. Currently, the iterators which can exploit available parallelism are:

- SGOPT optimizers - the genetic algorithm, coordinate pattern search, Solis-Wets, and stratified Monte Carlo methods within SGOPT.
- Parameter studies - DAKOTA’s `vector`, `list`, `centered`, and `multidim` parameter studies.

- Gradient-based optimizers - NPSOL, DOT, and OPT++ can all exploit parallelism through the use of DAKOTA's native finite differencing routine (selected with `method_source dakota` in the responses specification) which will perform concurrent evaluations whenever the model supports them.
- Speculative optimization - NPSOL, DOT, and OPT++ can speculate that the gradient information associated with a given line search point will be used later and compute the gradient information, either by finite difference or analytically, in parallel at the same time as the function values. This option is selected with the `speculative` keyword in the method specification and is used to balance the total amount of computation to be performed at each design point (allowing efficient utilization of multiple processors).

Single-processor DAKOTA specification

Specifying a single-processor DAKOTA job (which exploits parallelism through asynchronous calls to external job schedulers) requires inclusion of `asynchronous` in the interface specification. For example, the following specification runs an NPSOL optimization which will perform asynchronous finite differencing:

```
interface,                                              \
        application system,                             \
          asynchronous                                  \
          analysis_driver=  'qsub_script'

variables,                                              \
        continuous_design = 5                           \
          cdv_initial_point     0.2  0.05 0.08 0.2  0.2 \
          cdv_lower_bounds      0.15 0.02 0.05 0.1  0.1 \
          cdv_upper_bounds      1.0  1.0  1.0  1.0  1.0 \
          cdv_descriptor        'x1' 'x2' 'x3' 'x4' 'x5'

responses,                                              \
        num_objective_functions = 1                     \
        num_nonlinear_constraints = 2                   \
        numerical_gradients                             \
          interval_type central                         \
          method_source dakota                          \
          fd_step_size = 1.0E-4                         \
        no_hessians

method,                                                 \
        npsol_sqp
```

Note that `method_source dakota` is needed to invoke DAKOTA's internal finite differencing routine in order to exploit the parallelism. In this case, 11 function evaluations (one at the current point plus two deltas in each of five variables) can be performed simultaneously for each NPSOL response request. These 11 evaluations will be launched with system calls in the background and presumably assigned to additional processors through submission to a queue or similar approach.

Multiprocessor DAKOTA specification

Since the presence of multiple processors within the MPI context is detected automatically (whenever DAKOTA is launched in parallel with `mpirun` or `yod`), there is little to specify for the multiprocessor DAKOTA case. To run the same NPSOL example using the master-slave approach, `asynchronous` would be removed from the interface specification (since the slave servers execute their evaluations synchronously as described in Multiprocessor DAKOTA implementation on page 106):

```
interface,                                                  \
        application system,                                 \
          analysis_driver=  'qsub_script'
```

This will result in concurrent execution of single-processor evaluations managed by the self-scheduling master-slave algorithm.

If multilevel parallelism is being used, then `evaluation_servers` or `processors_per_evaluation` must additionally be specified to determine the processor partitioning to be used for a particular interface. In a more advanced example, a hybrid strategy which employs multilevel parallelism and which reconfigures the processor partitioning for varying model fidelity can be specified as follows:

```
strategy,                                                   \
        multi_level uncoupled                               \
        method_list = 'VPS', 'NLP'

variables,                                                  \
        continuous_design = 4                               \
          cdv_initial_point       1.0 1.0 1.0 1.0

method,                                                     \
        vector_parameter_study                              \
          id_method = 'VPS'                                 \
          step_vector = -.1 -.1 -.1 -.1                     \
          num_steps = 20                                    \
          interface_pointer = 'COARSE'                      \
          responses_pointer = 'NO_GRAD'

interface,                                                  \
        application direct,                                 \
          id_interface = 'COARSE'                           \
          analysis_driver = 'sim1'                          \
          processors_per_evaluation = 5

responses,                                                  \
        id_responses = 'NO_GRAD'                            \
        num_objective_functions = 1                         \
        num_nonlinear_constraints = 2                       \
        no_gradients                                        \
        no_hessians

method,                                                     \
        npsol_sqp                                           \
```

```
          id_method = 'NLP'                                  \
          interface_pointer = 'FINE'                         \
          responses_pointer = 'FD_GRAD'

interface,                                                   \
        application direct,                                  \
          id_interface = 'FINE'                              \
          analysis_driver = 'sim2'                           \
          processors_per_evaluation = 10

responses,                                                   \
        id_responses = 'FD_GRAD'                             \
        num_objective_functions = 1                          \
        num_nonlinear_constraints = 2                        \
        numerical_gradients                                  \
          interval_type central                              \
          method_source dakota                               \
          fd_step_size = 1.0E-4                              \
        no_hessians
```

If DAKOTA is executed on 40 processors (using `mpirun` or `yod`), then the study will first run a parameter study using a coarse model in which evaluations are scheduled through 8 servers of 5 processors each. The study will then pass the best parameter set to NPSOL which will perform parallel finite differencing (as in the previous examples) on a fine model using 4 servers of 10 processors each. Note that, for the multilevel parallel case, the `direct` application interface must be used for both interfaces (see Multiprocessor DAKOTA implementation on page 106).

Running a parallel DAKOTA job

Single-processor DAKOTA execution

Running a single-processor DAKOTA job (which exploits parallelism through asynchronous calls to external job schedulers) is identical to the procedure described in Running DAKOTA on page 123, e.g.:

```
dakota -i dakota.in > dakota.out
```

Multiprocessor DAKOTA execution

Running a multiprocessor DAKOTA job (which internally exploits parallelism) requires the use of an executable loading facility such as `mpirun` or `yod`.

On clusters of workstations, the `mpirun` script is used to initiate a parallel DAKOTA job, e.g.:

```
mpirun -np 12 dakota -i dakota.in > dakota.out
mpirun -machinefile machines -np 12 dakota -i dakota.in >
     dakota.out
```

where both examples specify the use of 12 processors, the former selecting them from a default system resources file and the latter specifying particular machines in a machine file (see [Gropp, W., and Lusk, E., 1996] for details).

On a massively parallel computer such as the TeraFLOPS machine, similar facilities are available from the Cougar operating system:

```
yod -sz 501 dakota -i dakota.in > dakota.out
```

In both the `mpirun` and `yod` cases, MPI command line arguments are used by MPI (extracted in the call to `MPI_Init`) and DAKOTA command line arguments are used by DAKOTA (extracted by DAKOTA's command line handler).

Caveats

MPI extracts its command line arguments first which can be problematic since certain file path specifications (e.g., "`../some_filename`") have been observed to cause problems, both for multiprocessor executions with `mpirun` and for single-processor executions of an executable configured with MPI (since `MPI_Init` is still called in this case). These path problems can be most easily resolved by using local linkage (all files or softlinks to the files appear in the same directory), which will likely be automated within a run script in a future software release.

Commands Introduction

Running DAKOTA on page 121 Restart Management on page 125

Overview

In the DAKOTA system, a *strategy* governs how each *method* maps *variables* into *responses* through the use of an *interface*. Each of these five pieces (*strategy*, *method*, *variables*, *responses*, and *interface*) are separate specifications in the user's input file, and as a whole, determine the study to be performed during an execution of the DAKOTA software. The number of strategies which can be invoked during a DAKOTA execution is limited to one. This strategy, however, may invoke multiple methods. Furthermore, each method may (in general) have its own "model," consisting of its own set of variables, its own set of responses, and its own interface. Thus, there may be multiple specifications of the *method*, *variables*, *responses*, and *interface* sections.

The syntax of DAKOTA specification is governed by the Input Deck Reader (IDR) parsing system [Weatherby, J.R., Schutt, J.A., Peery, J.S., and Hogan, R.E., 1996], which uses the `dakota.input.spec` file to describe the allowable inputs to the system. This input specification file provides a template of the allowable system inputs from which a particular input file (referred to herein as a `dakota.in` file) can be derived.

IDR Input Specification File

DAKOTA input is governed by the IDR input specification file. This file (`dakota.input.spec`) is used in the generation of parsing system files which are compiled into the DAKOTA executable. Therefore, `dakota.input.spec` is the *definitive source* for input syntax, capability options, and optional and required capability sub-parameters. Beginning users may find this file more confusing than helpful and, in this case, adaptation of example input files to a particular problem may be a more effective approach. However, advanced users can master all of the various input specification possibilities once the structure of the input specification file is understood. Key features include:

1. In the input specification, required parameters are enclosed in {}'s, optional parameters are enclosed in []'s, required groups are enclosed in ()'s, optional groups are enclosed in []'s, and either-or relationships are denoted by the | symbol. These symbols only appear in `dakota.input.spec`; they must not appear in actual user input files.
2. Keyword specifications (i.e., strategy, method, variables, interface, and responses) are delimited by newline characters. Therefore, to continue a keyword specification onto multiple lines, the back-slash character ("\") is needed to escape the newline. These newline escapes appear both in the input specification and in user input files.
3. Each of the five keyword specifications begins with a

```
<KEYWORD = name>, <FUNCTION = handler_name>
```

header which names the keyword and provides the binding to the keyword handler within DAKOTA's problem description database. In an actual input file, only the name of the keyword appears (e.g., `variables`).

4. Some of the specifications within a keyword indicate that the user must supply `<INTEGER>`, `<REAL>`, `<STRING>` or `<LISTof><INTEGER>`, `<LISTof><REAL>`, `<LISTof><STRING>` data as part of the specification. In an actual input file, the "=" is optional, `<LISTof>` data can be separated by commas or whitespace, and `<STRING>` data are enclosed in single quotes (e.g., `'text_book'`).

5. Input is order-independent (except for entries in data lists) and white-space insensitive. Although the order of input shown in the Sample dakota.in Files on page 118 generally follows the order of options in the input specification, this is not required.

6. Specifications may be abbreviated so long as the abbreviation is unique. For example, the `application` specification within the interface keyword could be abbreviated as `applic`, but should not be abbreviated as `app` since this would be ambiguous with `approximation`.

7. Comments are preceded by #.

The `dakota.input.spec` file used in DAKOTA V1.1 is:

```
<KEYWORD = variables>, <FUNCTION = variables_kwhandler>                    \
       [id_variables = <STRING>]                                           \
       [ {continuous_design = <INTEGER>}                                   \
             [cdv_initial_point = <LISTof><REAL>]                          \
             [cdv_lower_bounds = <LISTof><REAL>]                           \
             [cdv_upper_bounds = <LISTof><REAL>]                           \
             [cdv_descriptor = <LISTof><STRING>] ]                         \
       [ {discrete_design = <INTEGER>}                                     \
             [ddv_initial_point = <LISTof><INTEGER>]                       \
             [ddv_lower_bounds = <LISTof><INTEGER>]                        \
             [ddv_upper_bounds = <LISTof><INTEGER>]                        \
             [ddv_descriptor = <LISTof><STRING>] ]                         \
       [ {uncertain = <INTEGER>}                                           \
             [uv_descriptor = <LISTof><STRING>]                            \
             {uv_distribution_type = <LISTof><STRING>}                     \
             [uv_means = <LISTof><REAL>]                                   \
             [uv_std_deviations = <LISTof><REAL>]                          \
             [uv_lower_bounds = <LISTof><REAL>]                            \
             [uv_upper_bounds = <LISTof><REAL>]                            \
             [uv_filenames = <LISTof><STRING>] ]                           \
       [ {continuous_state = <INTEGER>}                                    \
             {csv_initial_state = <LISTof><REAL>}                          \
             [csv_descriptor = <LISTof><STRING>] ]                         \
       [ {discrete_state = <INTEGER>}                                      \
             {dsv_initial_state = <LISTof><INTEGER>}                       \
             [dsv_descriptor = <LISTof><STRING>] ]

<KEYWORD = responses>, <FUNCTION = responses_kwhandler>                    \
       [id_responses = <STRING>]                                           \
       [ {active_set_vector}  {on} | {off} ]                               \
       ( {num_objective_functions = <INTEGER>}                             \
         [num_nonlinear_constraints = <INTEGER>] )                         \
       |                                                                   \
       {num_least_squares_terms = <INTEGER>}                               \
       |                                                                   \
       {num_response_functions = <INTEGER>}                                \
       {no_gradients}                                                      \
```

```
        |                                                                       \
        ( {numerical_gradients}                                                 \
              [ {method_source} {dakota} | {vendor} ]                           \
              [ {interval_type} {forward} | {central} ]                         \
              [fd_step_size = <REAL>] )                                         \
        |                                                                       \
        {analytic_gradients}                                                    \
        |                                                                       \
        ( {mixed_gradients}                                                     \
              {id_numerical = <LISTof><INTEGER>}                                \
                [ {method_source} {dakota} | {vendor} ]                         \
                [ {interval_type} {forward} | {central} ]                       \
                [fd_step_size = <REAL>]                                         \
              {id_analytic = <LISTof><INTEGER>} )                               \
        {no_hessians}                                                           \
        |                                                                       \
        {analytic_hessians}

  <KEYWORD = interface>, <FUNCTION = interface_kwhandler>                       \
        [id_interface = <STRING>]                                               \
        ( {application}                                                         \
              ( {analysis_driver = <STRING>}                                    \
                [input_filter = <STRING>]                                       \
                [output_filter = <STRING>] )                                    \
              |                                                                 \
              ( {concurrent_drivers = <LISTof><STRING>}                         \
                [pre_driver = <STRING>]                                         \
                [post_driver = <STRING>] )                                      \
              ( {system} [asynchronous]                                         \
                [parameters_file = <STRING>]                                    \
                [results_file = <STRING>]                                       \
                [analysis_usage = <STRING>]                                     \
                [aprepro]                                                       \
                [file_tag]                                                      \
                [file_save] )                                                   \
              |                                                                 \
              ( {direct} [asynchronous]                                         \
                [evaluation_servers = <INTEGER>]                                \
                [processors_per_evaluation = <INTEGER>] )                       \
              [ {failure_capture} {abort} | {retry = <INTEGER>} |               \
                {recover = <LISTof><REAL>} | {continuation} ] )                 \
        |                                                                       \
        ( {approximation}                                                       \
              {neural_network} | {response_surface} |                           \
                 {multi_point}    | {mars_surface} )                          \
        |                                                                       \
        {test = <STRING>}

  <KEYWORD = strategy>, <FUNCTION = strategy_kwhandler>                         \
        ( {multi_level}                                                         \
            ( {uncoupled}                                                       \
              [ {adaptive} {progress_threshold = <REAL>} ]                      \
              {method_list = <LISTof><STRING>} )                                \
            |                                                                   \
            ( {coupled}                                                         \
              {global_method = <STRING>}                                        \
              {local_method = <STRING>}                                         \
              [local_search_probability = <REAL>] ) )                           \
        |                                                                       \
        ( {seq_approximate_opt}                                                 \
              {opt_method = <STRING>}                                           \
              {approximate_interface = <STRING>}                                \
              {actual_interface = <STRING>} )                                   \
        |                                                                       \
        ( {opt_under_uncertainty}                                               \
              {opt_method = <STRING>}                                           \
              {nond_method = <STRING>} )                                        \
        |                                                                       \
        ( {branch_and_bound}                                                    \
              {opt_method = <STRING>}                                           \
              {iterator_servers = <INTEGER>} )                                  \
        |                                                                       \
```

```
        ( {single_method}                                                   \
              [method_pointer = <STRING>] )

      <KEYWORD = method>, <FUNCTION = method_kwhandler>                     \
        [id_method = <STRING>]                                              \
        [interface_pointer = <STRING>]                                      \
        [variables_pointer= <STRING>]                                       \
        [responses_pointer = <STRING>]                                      \
        [speculative]                                                       \
        [ {output} {verbose} | {quiet} ]                                    \
        [linear_constraints = <LISTof><REAL>]                               \
        [max_iterations = <INTEGER>]                                        \
        [max_function_evaluations = <INTEGER>]                              \
        [constraint_tolerance = <REAL>]                                     \
        [convergence_tolerance = <REAL>]                                    \
        ( {dot_frcg}                                                        \
              [ {optimization_type} {minimize} | {maximize} ] )             \
        |                                                                   \
        ( {dot_mmfd}                                                        \
              [ {optimization_type} {minimize} | {maximize} ] )             \
        |                                                                   \
        ( {dot_bfgs}                                                        \
              [ {optimization_type} {minimize} | {maximize} ] )             \
        |                                                                   \
        ( {dot_slp}                                                         \
              [ {optimization_type} {minimize} | {maximize} ] )             \
        |                                                                   \
        ( {dot_sqp}                                                         \
              [ {optimization_type} {minimize} | {maximize} ] )             \
        |                                                                   \
        ( {npsol_sqp}                                                       \
              [verify_level = <INTEGER>]                                    \
              [function_precision = <REAL>]                                 \
              [linesearch_tolerance = <REAL>] )                             \
        |                                                                   \
        ( {optpp_cg}                                                        \
              [max_step = <REAL>]                                           \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_q_newton}                                                  \
              [ {search_method} {value_based_line_search} |                 \
                {gradient_based_line_search} | {trust_region} ]             \
              [max_step = <REAL>]                                           \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_g_newton}                                                  \
              [ {search_method} {value_based_line_search} |                 \
                {gradient_based_line_search} | {trust_region} ]             \
              [max_step = <REAL>]                                           \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_newton}                                                    \
              [ {search_method} {value_based_line_search} |                 \
                {gradient_based_line_search} | {trust_region} ]             \
              [max_step = <REAL>]                                           \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_fd_newton}                                                 \
              [ {search_method} {value_based_line_search} |                 \
                {gradient_based_line_search} | {trust_region} ]             \
              [max_step = <REAL>]                                           \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_baq_newton}                                                \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_ba_newton}                                                 \
              [gradient_tolerance = <REAL>] )                               \
        |                                                                   \
        ( {optpp_bcq_newton}                                                \
              [ {search_method} {value_based_line_search} |                 \
                {gradient_based_line_search} | {trust_region} ]             \
```

```
      [max_step = <REAL>]                                                \
      [gradient_tolerance = <REAL>] )                                    \
|                                                                        \
( {optpp_bcg_newton}                                                     \
      [ {search_method} {value_based_line_search} |                      \
        {gradient_based_line_search} | {trust_region} ]                  \
      [max_step = <REAL>]                                                \
      [gradient_tolerance = <REAL>] )                                    \
|                                                                        \
( {optpp_bc_newton}                                                      \
      [ {search_method} {value_based_line_search} |                      \
        {gradient_based_line_search} | {trust_region} ]                  \
      [max_step = <REAL>]                                                \
      [gradient_tolerance = <REAL>] )                                    \
|                                                                        \
( {optpp_bc_ellipsoid}                                                   \
      [initial_radius = <REAL>]                                          \
      [gradient_tolerance = <REAL>] )                                    \
|                                                                        \
( {optpp_pds}                                                            \
      [search_scheme_size = <INTEGER>] )                                 \
|                                                                        \
{optpp_test_new}                                                         \
|                                                                        \
( {sgopt_pga_real}                                                       \
      [solution_accuracy = <REAL>]                                       \
      [max_cpu_time = <REAL>]                                            \
      [seed = <INTEGER>]                                                 \
      [population_size = <INTEGER>]                                      \
      [ {selection_pressure} {rank = <REAL>} |                           \
        {proportional} ]                                                 \
      [ {replacement_type} {random = <INTEGER>} |                        \
        {CHC = <INTEGER>} | {elitist = <INTEGER>}                        \
        [new_solutions_generated = <INTEGER>] ]                          \
      [ {crossover_type} {two_point} | {mid_point} |                     \
        {blend} | {uniform}                                              \
        [crossover_rate = <REAL>] ]                                      \
      [ {mutation_type} ( {normal} [std_deviation = <REAL>] )            \
                        | {interval} | {cauchy}                          \
        [dimension_rate = <REAL>]                                        \
        [population_rate = <REAL>] ] )                                   \
|                                                                        \
( {sgopt_pga_int}                                                        \
      [solution_accuracy = <REAL>]                                       \
      [max_cpu_time = <REAL>]                                            \
      [seed = <INTEGER>]                                                 \
      [population_size = <INTEGER>]                                      \
      [ {selection_pressure} {rank = <REAL>} |                           \
        {proportional} ]                                                 \
      [ {replacement_type} {random = <INTEGER>} |                        \
        {CHC = <INTEGER>} | {elitist = <INTEGER>}                        \
        [new_solutions_generated = <INTEGER>] ]                          \
      [ {crossover_type} {two_point} | {uniform}                         \
        [crossover_rate = <REAL>] ]                                      \
      [ {mutation_type} {offset} | {interval}                            \
        [dimension_rate = <REAL>]                                        \
        [population_rate = <REAL>] ] )                                   \
|                                                                        \
( {sgopt_coord_ps}                                                       \
      [solution_accuracy = <REAL>]                                       \
      [max_cpu_time = <REAL>]                                            \
      [ {expansion_policy} {unlimited} | {once} ]                        \
      [expand_after_success = <INTEGER>]                                 \
      [expansion_exponent = <INTEGER>]                                   \
      [contraction_exponent = <INTEGER>]                                 \
      {initial_delta = <REAL>}                                           \
      {threshold_delta = <REAL>}                                         \
      [ {exploratory_moves} {standard} | {offset} |                      \
        {best_first} | {biased_best_first} ] )                           \
|                                                                        \
( {sgopt_coord_sps}                                                      \
      [solution_accuracy = <REAL>]                                       \
```

```
      [max_cpu_time = <REAL>]                                          \
      [seed = <INTEGER>]                                               \
      [ {expansion_policy} {unlimited} | {once} ]                      \
      [expand_after_success = <INTEGER>]                               \
      [expansion_exponent = <INTEGER>]                                 \
      [contraction_exponent = <INTEGER>]                               \
      {initial_delta = <REAL>}                                         \
      {threshold_delta = <REAL>}                                       \
      [ {exploratory_moves} {standard} | {offset} |                    \
        {best_first} | {biased_best_first} ] )                         \
|                                                                      \
( {sgopt_solis_wets}                                                   \
      [solution_accuracy = <REAL>]                                     \
      [max_cpu_time = <REAL>]                                          \
      [seed = <INTEGER>]                                               \
      [expand_after_success = <INTEGER>]                               \
      [contract_after_failure = <INTEGER>]                             \
      [initial_rho = <REAL>]                                           \
      [threshold_rho = <REAL>] )                                       \
|                                                                      \
( {sgopt_strat_mc}                                                     \
      [solution_accuracy = <REAL>]                                     \
      [max_cpu_time = <REAL>]                                          \
      [seed = <INTEGER>]                                               \
      [partitions = <LISTof><INTEGER>] )                               \
|                                                                      \
( {nond_probability}                                                   \
      {observations = <INTEGER>}                                       \
      [seed = <INTEGER>]                                               \
      {sample_type} {random} | {lhs}                                   \
      {response_thresholds = <LISTof><REAL>} )                         \
|                                                                      \
( {nond_mean_value}                                                    \
      {response_filenames = <LISTof><STRING>} )                        \
|                                                                      \
( {vector_parameter_study}                                             \
      ( {final_point = <LISTof><REAL>}                                 \
        {step_length = <REAL>} | {num_steps = <INTEGER>} )             \
      |                                                                \
      ( {step_vector = <LISTof><REAL>}                                 \
        {num_steps = <INTEGER>} ) )                                    \
|                                                                      \
( {list_parameter_study}                                               \
      {list_of_points = <LISTof><REAL>} )                              \
|                                                                      \
( {centered_parameter_study}                                           \
      {percent_delta = <REAL>}                                         \
      {deltas_per_variable = <INTEGER>} )                              \
|                                                                      \
( {multidim_parameter_study}                                           \
      {partitions = <LISTof><INTEGER>} )
```

In the variables keyword, the main structure is that of the five optional group specifications for continuous design, discrete design, uncertain, continuous state, and discrete state variables. Each of these specifications can either appear or not appear as a group. Within the responses keyword, the primary structure is the required specification of the function set (either an optimization function set OR a least squares function set OR a generic function set must appear), followed by the required specification of the gradients (either none OR numerical OR analytic OR mixed must be specified) followed by the required specification of the Hessians (either none OR analytic must be specified). Next, the interface keyword requires the selection of either an application OR an approximation OR a test interface. Within the application block, the type must be specified with either the system OR the direct required group specification. The strategy specification is relatively simple, requiring either a multilevel OR a sequential approximate

optimization OR an optimization under uncertainty OR a branch and bound OR a single method strategy specification. Within the multilevel group specification, either an uncoupled OR a coupled group specification must be supplied. Lastly, the method keyword is the most involved specification; however, its structure is relatively simple. The structure is simply that of a sequence of optional method-independent settings followed by a long list of possible methods appearing as required group specifications (containing a variety of method-dependent settings) separated by OR's. Refer to Variables Commands on page 134, Responses Commands on page 141, Interface Commands on page 127, Strategy Commands on page 150, and Method Commands on page 156 for detailed information on the keywords and their various optional and required specifications. And for additional details on IDR specification logic and rules, refer to [Weatherby, J.R., Schutt, J.A., Peery, J.S., and Hogan, R.E., 1996].

Common Specification Mistakes

Spelling and omission of required parameters are the most common errors. Less obvious errors include:

1. Documentation of new capability can lag the use of new capability in executables. When parsing errors occur which the documentation cannot explain, reference to the particular input specification used in building the executable will often resolve the errors.
2. Since keywords are terminated with the newline character, care must be taken to avoid following the backslash character with any white space since the newline character will not be properly escaped, resulting in parsing errors due to the truncation of the keyword specification.
3. Care must be taken to include newline escapes when embedding comments within a keyword specification. That is, newline characters will signal the end of a keyword specification even if they are part of a comment line. For example, the following specification will be truncated because the embedded comment neglects to escape the newline:

```
# No error here: newline need not be escaped since comment is not embedded
responses,                                      \
        num_objective_functions = 1             \
# Error here: this comment must escape the newline
        analytic_gradients                      \
        no_hessians
```

In most cases, the IDR system provides helpful error messages which will help the user isolate the source of the parsing problem.

Sample dakota.in Files

A DAKOTA input file is a collection of the fields allowed in the `dakota.input.spec` specification file which describe the problem to be solved by the DAKOTA system. Several examples follow.

Sample 1: Optimization

The following sample input file shows single-method optimization of the Textbook Example on page 192 using DOT's modified method of feasible directions. It is available in the test directory as `Dakota/test/dakota_textbook.in`. Helpful notes are included in this sample input file as comments.

```
# DAKOTA INPUT FILE - dakota_textbook.in
# NOTES: Specifications are delimited by newline characters. Therefore, to
#        continue a specification onto multiple lines, the back-slash character
#        is needed to escape the newline. Input is order-independent and
#        white-space insensitive. Keywords may be abbreviated so long as the
#        abbreviation is unique. Comments are preceded by #. Helpful NOTES
#        precede each section specification; however, the definitive resources
#        for input grammar are Dakota/src/dakota.input.spec and the Commands
#        chapter of the User's Instructions manual.

# Interface section specification
# NOTES: Interfaces are 1 of 3 main types: application interfaces are used for
#        interfacing with simulation codes, approximation interfaces use
#        inexpensive design space approximations in place of expensive
#        simulations, and test interfaces use linked-in test functions for
#        algorithm testing purposes (to eliminate system call overhead).
#        Application interfaces can be further categorized into system and
#        direct types. The system type uses system calls to invoke the
#        simulation, while the direct type uses the same constructs as the test
#        interface for linked-in simulation codes.  Both application interface
#        types use analysis_driver, input_filter, and output_filter
#        specifications. The system type additionally uses parameters_file,
#        results_file, analysis_usage, aprepro, file_tag, and file_save
#        specifications. The analysis_driver provides the name of the analysis
#        executable, driver script, or linked module; the input_filter and
#        output_filter provide pre- and post-processing for the analysis in the
#        procedure of mapping parameters into responses (default = NO_FILTER);
#        the parameters_file and results_file are data files which Dakota
#        creates and reads, respectively, in the system call case (default =
#        Unix temp files); analysis_usage defines nontrivial command syntax
#        (default = standard syntax); aprepro controls the format of the
#        parameters file for usage with the APREPRO utility; file_tag controls
#        the unique tagging of data files with function evaluation number
#        (default = no tagging); and file_save controls whether or not file
#        cleanup operations are performed (default = data files are removed
#        when no longer in use). Most settings are optional with meaningful
#        defaults as shown above. Refer to the Interface Commands section in
#        the User's Instructions manual for additional information.
interface,                                                              \
       application system,                                              \
         input_filter    =       'NO_FILTER'                           \
         output_filter   =       'NO_FILTER'                           \
         analysis_driver =       'text_book'                           \
         parameters_file =       'text_book.in'                        \
         results_file    =       'text_book.out'                       \
         analysis_usage  =       'DEFAULT'                             \
         file_tag                                                      \
         file_save

# Variables specification
# NOTES: A variables set can contain design, uncertain, and state variables
#        for continuous, discrete, or mixed variable problem domains.
#        Design variables are those variables which an optimizer adjusts in
#        order to locate an optimal design.  Each of the design parameters
#        can have an initial point, a lower bound, an upper bound, and a
#        descriptive tag.  Uncertain variables are those variables which are
#        characterized by probability distributions.  Each uncertain variable
#        specification can contain a distribution type, a mean, a standard
#        deviation, a lower bound, an upper bound, a histogram filename and a
#        descriptive tag.  State variables are "other" variables which are to
#        be mapped through the interface.  Each state variable specification
#        can have an initial state and a descriptor.  State variables provide a
```

```
#        convenience mechanism for parameterizing additional model inputs, such
#        as mesh density, solver convergence tolerance and time step controls,
#        and will be used to enact model adaptivity in future strategy
#        developments.

variables,                                                                    \
       continuous_design = 2                                                  \
         cdv_initial_point    0.9    1.1                                     \
         cdv_upper_bounds     5.8    2.9                                     \
         cdv_lower_bounds     0.5   -2.9                                     \
         cdv_descriptor       'x1'   'x2'

# Responses specification
# NOTES: This specification implements a generalized Dakota data set by
#        specifying a set of functions and the types of gradients and Hessians
#        for these functions. Optimization data sets require specification of
#        num_objective_functions and num_nonlinear_constraints. Multiobjective
#        opimization is not yet supported, so num_objective_functions must
#        currently be equal to 1. Uncertainty quantification data sets are
#        specified by num_response_functions. Nonlinear least squares data
#        sets are specified with num_least_squares_terms. Gradient type
#        specification may be no_gradients, analytic_gradients,
#        numerical_gradients or mixed_gradients. Numerical and mixed gradient
#        specifications can optionally include selections for method_source,
#        interval_type, and fd_step_size. Mixed_gradient specifications require
#        id_numerical & id_analytic lists to specify the gradient types for
#        different function numbers. Hessian type specification may currently
#        be no_hessians or analytic_hessians.

responses,                                                                    \
       num_objective_functions = 1                                            \
       num_nonlinear_constraints = 2                                          \
       analytic_gradients                                                     \
       no_hessians

# Strategy specification
# NOTES: Contains specifications for multilevel, SAO, and OUU strategies.  The
#        single_method strategy is a "fall through" strategy, in that in only
#        invokes a single iterator.  If no strategy specification appears, then
#        single_method is the default.

strategy,                                                                     \
       single_method

# Method specification
# NOTES: method can currently be dot_frcg, dot_mmfd, dot_bfgs, dot_slp,
#        dot_sqp, npsol_sqp, optpp_cg, optpp_q_newton, optpp_g_newton,
#        optpp_newton, optpp_fd_newton, optpp_ba_newton, optpp_baq_newton,
#        optpp_bc_newton, optpp_bcq_newton, optpp_bcg_newton,
#        optpp_bc_ellipsoid, optpp_pds, optpp_test_new, sgopt_pga_real,
#        sgopt_pga_int, sgopt_coord_ps, sgopt_coord_sps, sgopt_solis_wets,
#        sgopt_strat_mc, nond_probability, nond_mean_value,
#        vector_parameter_study, list_parameter_study,
#        centered_parameter_study, or multidim_parameter_study. Most method
#        control parameters are optional with meaningful defaults. Default
#        values for optional parameters are defined in the DataMethod class
#        constructor and are documented in the Method Commands section of the
#        User's Instructions manual.

method,                                                                       \
        dot_mmfd,                                                             \
         max_iterations = 50,                                                 \
         convergence_tolerance = 1e-4                                         \
         output verbose                                                       \
         optimization_type minimize
```

Sample 2: Least Squares

The following sample input file shows a nonlinear least squares solution of the Rosenbrock Example on page 204 using OPT++'s Gauss-Newton method. It is available in the test directory as `Dakota/test/dakota_rosenbrock.in`.

```
interface,                                                              \
       application system,                                              \
         analysis_driver =    'rosenbrock_ls'

variables,                                                              \
       continuous_design = 2                                            \
        cdv_initial_point            -1.2 1.0                           \
        cdv_lower_bounds             -2.0 -2.0                          \
        cdv_upper_bounds              2.0 2.0                           \
        cdv_descriptor                'x1' 'x2'

responses,                                                              \
       num_least_squares_terms = 2                                      \
       analytic_gradients                                               \
       no_hessians

method,                                                                 \
       optpp_bcg_newton,                                                \
       max_iterations = 50,                                             \
       convergence_tolerance = 1e-4
```

Sample 3: Nondeterministic Analysis

The following sample input file shows Latin Hypercube Monte Carlo sampling using the Textbook Example on page 192. It is available in the test directory as `Dakota/test/dakota_textbook_lhs.in`.

```
interface,                                                              \
       application system,                                              \
         analysis_driver=        'text_book'

variables,                                                              \
       uncertain = 2                                                    \
         uv_distribution_type = 'normal' 'normal'                       \
         uv_means             =  248.89, 593.33                         \
         uv_std_deviations    =   12.4,   29.7                          \
         uv_lower_bounds      =  199.3,  474.63                         \
         uv_upper_bounds      =  298.5,  712.                           \
         uv_descriptor        =  'TF1'   'TF2'

responses,                                                              \
       num_response_functions = 3                                       \
       no_gradients                                                     \
       no_hessians

strategy,                                                               \
       single_method

method,                                                                 \
        nond_probability,                                               \
          observations = 20,                                            \
          response_thresholds = 1.2e+11 6.e+04 3.5e+05\
         seed = 1                                                       \
         sample_type lhs
```

Sample 4: Parameter Study

The following sample input file shows a 1-D vector parameter study using the Textbook Example on page 192. It is available in the test directory as `Dakota/test/dakota_pstudy.in`.

```
interface,                                                              \
       application system,                                              \
         asynchronous                                                   \
         analysis_driver = 'text_book'

variables,                                                              \
       continuous_design = 3                                            \
         cdv_initial_point       1.0 1.0 1.0

responses,                                                              \
       num_objective_functions = 1                                      \
       num_nonlinear_constraints = 2                                    \
       analytic_gradients                                               \
       analytic_hessians

method,                                                                 \
        vector_parameter_study                                          \
         step_vector = .1 .1 .1                                         \
         num_steps = 4
```

Sample 5: Multilevel Hybrid Strategy

The following sample input file shows a multilevel hybrid strategy using three iterators. It employs a genetic algorithm, coordinate pattern search and full Newton gradient-based optimization in succession to solve the Textbook Example on page 192. It is available in the test directory as `Dakota/test/dakota_multilevel.in`.

```
strategy,                                                               \
       multi_level uncoupled                                            \
         method_list = 'GA' 'CPS' 'NLP'

method,                                                                 \
       sgopt_pga_real                                                   \
         id_method = 'GA'                                               \
         variables_pointer = 'V1'                                       \
         interface_pointer = 'I1'                                       \
         responses_pointer = 'R1'                                       \
         population_size = 10                                           \
         verbose output

method,                                                                 \
       sgopt_coord_sps                                                  \
         id_method = 'CPS'                                              \
         variables_pointer = 'V1'                                       \
         interface_pointer = 'I1'                                       \
         responses_pointer = 'R1'                                       \
         verbose output                                                 \
         initial_delta = 0.1                                            \
         threshold_delta = 1.e-4                                        \
         solution_accuracy = 1.e-10                                     \
         exploratory_moves best_first

method,                                                                 \
        optpp_newton                                                    \
         id_method = 'NLP'                                              \
         variables_pointer = 'V1'                                       \
         interface_pointer = 'I1'                                       \
         responses_pointer = 'R2'                                       \
         gradient_tolerance = 1.e-12                                    \
         convergence_tolerance = 1.e-15
```

```
interface,                                                              \
       id_interface = ‘I1’                                              \
       application direct,                                              \
         analysis_driver=  ‘text_book’

variables,                                                              \
       id_variables = ‘V1’                                              \
       continuous_design = 2                                            \
         cdv_initial_point    0.6    0.7\
         cdv_upper_bounds     5.8    2.9      \
         cdv_lower_bounds     0.5   -2.9      \
         cdv_descriptor       ‘x1’   ‘x2’

responses,                                                              \
       id_responses = ‘R1’                                              \
       num_objective_functions = 1                                      \
       no_gradients                                                     \
       no_hessians

responses,                                                              \
       id_responses = ‘R2’                                              \
       num_objective_functions = 1                                      \
       analytic_gradients                                               \
       analytic_hessians
```

Running DAKOTA

Basic information required for running DAKOTA includes the name and location of the executable program and the command line syntax and options.

Executable Location

Remote installations

After installing and building the system from a new code distribution (see Distributions and Checkouts on page 180 and Basic Installation on page 180), the DAKOTA executable will reside in `Dakota/src/<canonical_build_dir>/dakota`, where the canonical name describes the platform and operating system under which the executable was built (e.g., `sparc-sun-solaris2.5.1`). The `dakota` file in the `Dakota/test` directory is a soft link to the `Dakota/src/<canonical_build_dir>/dakota` executable.

Sandia developer-supported installations

The DAKOTA executable will have already been built by the DAKOTA developers and installed in `/usr/local/bin` on the supported server machines. For file systems shared by multiple platforms, simplified canonical names are sometimes used to distinguish between the executables (e.g., `dakota_sun`, `dakota_hp`, `dakota_sgi`, `dakota_ibm`, etc.). For file systems unique to a single platform (as is generally the case with `/usr/local/bin`), `dakota` without any canonical modifiers is used.

For the following discussions, it will be assumed that an executable named `dakota` is available in the user's path.

Command Line Inputs

Executing the program with the following syntax:

```
dakota
```

will result in the following usage message which describes the various optional and required command line inputs:

```
usage: dakota [options and <args>]

       -help (Print this summary)
       -input <$val> (REQUIRED Dakota Problem Description file $val)
       -read_restart <$val> (Read a previously written Dakota restart log file
$val)
       -stop_restart <$val> (Stop restart file processing at evaluation $val)
       -write_restart [$val] (Write a new Dakota restart log file $val)
```

Of these available command line inputs, only the "`-input`" option is required. The command line input parser implemented in the **CommandLineHandler** class allows abbreviation so long as the abbreviation is unique. For example "`-i`" is commonly used in place of "`-input`."

The "`-help`" option prints the usage message above. The "`-input`" option provides the name of the DAKOTA input file (see Sample dakota.in Files on page 118 for examples). The "`-read_restart`" and "`-write_restart`" command line inputs provide the names of restart databases to read from and write to, respectively. The "`-stop_restart`" command line input limits the number of function evaluations read from the restart database (the default is all the evaluations) for those cases in which some evaluations were erroneous or corrupted.

Execution Syntax

Input/Output Management

To run DAKOTA with a particular input file, the following syntax can be used:

```
dakota -i dakota.in
```

This will echo stdout and stderr to the terminal. To redirect output to a file, any of a variety of redirection variants can be used. The simplest of these redirects stdout:

```
dakota -i dakota.in > dakota.out
```

To append to a file rather than overwrite it, ">>" is used in place of ">". To redirect stderr as well as stdout, a "&" is appended with no embedded space, i.e. ">&" or ">>&" is used. To override the noclobber environment variable (if set) in order to allow overwriting of an existing output file or appending of a file that does not yet exist, a "!" is appended with no embedded space, i.e. ">!", ">&!", ">>!", or ">>&!" is used.

To run the dakota process in the background, append an ampersand to the command with an embedded space, e.g.:

```
dakota -i dakota.in > dakota.out &
```

Refer to [Anderson, G., and Anderson, P., 1986] for more information on redirection and background commands.

Restart Management

To write a restart file using a particular name, the `-write_restart` command line input is used:

```
dakota -i dakota.in -write_restart my_restart_file
```

If no `-write_restart` specification is used, then DAKOTA will write a restart file using the default name `dakota.rst`.

To restart DAKOTA from a restart file, the `-read_restart` command line input is used:

```
dakota -i dakota.in -read_restart my_restart_file
```

If no `-read_restart` specification is used, then DAKOTA will not read restart information from any file (i.e., the default is no restart processing).

If the `-write_restart` and `-read_restart` specifications identify the same file (including the case where `-write_restart` is not specified and `-read_restart` identifies `dakota.rst`), then new evaluations will be appended to the existing restart file. If the `-write_restart` and `-read_restart` specifications identify different files, then the evaluations read from the file identified by `-read_restart` are first written to the `-write_restart` file. Any new evaluations are then appended to the `-write_restart` file. In this way, restart operations can be chained together indefinitely with the assurance that all of the relevant evaluations are present in the latest restart file.

To read in only a portion of a restart file, the `-stop_restart` control is used. Note that the integer value specified refers to the number of entries to be read from the database, which may differ from the evaluation number in the previous run if any duplicates were detected (since duplicates are not replicated in the restart file). In the case of a `-stop_restart` specification, it is usually desirable to specify a new restart file using `-write_restart` so as to remove the records of erroneous or corrupted function evaluations. For example, to read in the first 50 evaluations from `dakota.rst`:

```
dakota -i dakota.in -read_restart dakota.rst
     -stop_restart 50 -write_restart dakota_new.rst
```

The `dakota_new.rst` file will contain the 50 processed evaluations from `dakota.rst` as well as any new evaluations. All evaluations following the 50^{th} in `dakota.rst` have been removed from the latest restart record.

DAKOTA's restart algorithm relies on its duplicate detection capabilities. Processing a restart file populates the list of function evaluations that have been performed. Then, when the study is reinitiated, many of the function evaluations requested by the iterator are intercepted by the duplicate detection code. This approach has the primary advantage of restoring the complete state of the iteration (including the ability to correctly detect subsequent duplicates) for all

iterators and multi-iterator strategies without the need for iterator-specific restart code. However, the possibility exists for numerical round-off error to cause a divergence between the evaluations performed in the previous and restarted studies. This has been extremely rare to date.

Tabular descriptions

In the following discussions of keyword specifications, tabular formats (Table 9 through Table 56) are used to present a short description of the specification, the actual syntax of the specification from `dakota.input.spec`, a sample specification as it would appear in an input file, the status of the specification (required, optional, required group, or optional group), and the default for an optional specification.

It can be difficult to capture in a simple tabular format the complex relationships that can occur when specifications are nested within multiple groupings. For example, in an interface keyword, the `parameters_file` specification is an optional specification within a required group specification (`system`) separated from another required group specification (`direct`) by a logical OR. The selection of `system` or `direct` is contained within another required group specification (`application`) separated from other required group specifications (`approximation`, `test`) by logical OR's. Thus, concisely describing a specification status in a table fails to capture the complete picture of the specification inter-relationships which are present in `dakota.input.spec`.

To better capture these relationships, this documentation presents the various group specifications in separate tables. Details of the outermost required groups are presented in one or more tables (e.g., `application` versus `approximation` versus `test` in Table 10, Table 13, and Table 14), and details of each of the innermost required groups are presented in additional tables (e.g., `system` versus `direct` in Table 11 and Table 12). Ellipsis (...) are used within tabular entries for group specifications to denote omissions from the group specification which are explained in subsequent table entries.

Interface Commands

Description

The interface section in a DAKOTA input file specifies how function evaluations will be performed. The three mechanisms currently in place for performing function evaluations involve interfacing with a simulation, an approximation, or a test function. In the former case of a simulation, the `application` interface is used to invoke the simulation with either system calls or direct function calls. In the system call case, communication between DAKOTA and the simulation occurs through parameter and response files, and in the direct function case, communication occurs through the function parameter list. More information and examples on interfacing with simulations is provided in The Application Interface on page 79. In the case of an approximation, an `approximation` interface can be selected to make use of surrogate modeling capabilities available within DAKOTA's **ApproximationInterface** class hierarchy (see The Approximation Interface on page 95). Lastly, a `test` interface can be selected for direct access to polynomial test functions which are compiled into the DAKOTA executable as part of the direct function capability (see The Direct Function Application Interface on page 80). The `test` interface provides a means for testing algorithms and strategies without system call overhead and without the expense of engineering simulations.

Several examples follow. The first example shows an application interface specification which specifies the use of system calls, the names of the analysis executable and the parameters and results files, that separate filters will not be used, that no special analysis usage syntax will be specified, and that parameters and responses files will be tagged and saved. Refer to The Application Interface on page 79 for more information on the use of these options.

```
interface,\
  application system,\
    input_filter    =       'NO_FILTER'\
    output_filter   =       'NO_FILTER'\
    analysis_driver =       'rosenbrock'\
    parameters_file =       'params.in'\
    results_file    =       'results.out'\
    analysis_usage  =       'DEFAULT'\
    file_tag\
    file_save
```

The next example shows an approximation interface specification which invokes the response surface approximation methodology.

```
interface,\
  approximation,\
    response_surface
```

The next example shows an test interface specification which specifies use of the `text_book` internal test function.

```
interface,\
  test = 'text_book'
```

Specification

The interface specification has the following structure:

```
interface,\
   <set identifier>\
   <application specification>
      or <approximation specification>
      or <test specification>
```

Referring to the IDR Input Specification File on page 112, it is evident from the brackets that the set identifier is an optional specification, and from the three required groups (enclosing in parentheses) separated by OR's, that one and only one of the three interface specifications (`application`, `approximation`, or `test`) must be provided.

The optional set identifier can be used to provide a unique identifier string for labeling a particular interface specification. A method can then identify the use of a particular interface by specifying this label in its `interface_pointer` specification (see Method Commands on page 156). The application, approximation, or test specification is used to define the specifics of the interface to be used by a method for the mapping of parameters into responses. The following sections describe each of these interface specifications in additional detail.

Developer's notes: In the C++ implementation, the different interface classes are part of the **DakotaInterface** class hierarchy which uses the virtual `map` function to polymorphically define the interface's functionality. This allows the specific identity and complexities of the interface to be hidden from the method since the use of the `map` functionality is common among all interfaces.

Set Identifier

The optional set identifier specification uses the keyword `id_interface` to input a string for use in identifying a particular interface specification with a particular method (see also `interface_pointer` in the Method Commands on page 156). For example, a method whose specification contains `interface_pointer = 'I1'` will use an interface specification with `id_interface = 'I1'`.

It is appropriate to omit an `id_interface` string in the interface specification and a corresponding `interface_pointer` string in the method specification if only one interface specification is included in the input file, since the binding of a method to an interface is unambiguous in this case. More specifically, if a method omits specifying an `interface_pointer`, then it will use the last interface specification parsed, which has the least potential for confusion when only a single interface specification exists. Table 9 summarizes the set identifier inputs.

Table 9 Specification detail for set identifier

Description	Specification	Sample	Status	Default
Interface set identifier	[id_interface = <STRING>]	id_interface = ‘I1’	Optional	use of last interface parsed

Application Interface

The application interface uses a simulator program, and optionally filter programs, to perform the parameter to response mapping. The simulator and filter programs are invoked with either system calls or direct function calls. In the former case, files are used for transfer of parameter and response data between DAKOTA and the simulator program. This approach is simple and reliable and does not require any modification to simulator programs. In the latter direct function case, the function parameter list is used to pass data. This approach requires modification to simulator programs so that they can be linked into DAKOTA; however it can be more efficient through the elimination of system call overhead, can be less prone to loss of precision in that data can be passed directly rather than written to and read from a file, and can enable multilevel parallelism through MPI communicator partitioning as described in Implementation of Parallelism on page 104.

The application interface group specification contains several specifications which are valid for all application interfaces as well as additional specifications pertaining specifically to system call and direct application interfaces. Table 10 summarizes the specifications valid for all application interfaces, and Table 11 and Table 12 summarize the additional specifications for system call and direct application interfaces. In Table 10, the names of the input filter, output filter, and analysis driver executables are supplied as strings using the `input_filter`, `output_filter`, and `analysis_driver` specifications. Both the system call and direct function application interfaces use these same specifications. The `analysis_driver` specification is required, and the `input_filter` and `output_filter` specifications are optional with the default behavior of no filter usage (string default is `'NO_FILTER'`). If no filters are used, then the interface is called a “1-piece Interface”; if filters are used, it is called a “3-piece Interface.” Failure capturing in application interfaces is governed by the `failure_capture` specification. Supported directives for mitigating captured failures, as described in Failure capturing on page 93, are `abort`, `retry`, `recover`, and `continuation`.

Table 10 Specification detail for application interfaces

Description	Specification	Sample	Status	Default
Application interface	({application} ...)	application	Required group	N/A
Input filter	[input_filter = <STRING>]	input_filter = ‘ifilter.exe’	Optional	no input filter

Table 10 Specification detail for application interfaces

Description	Specification	Sample	Status	Default
Output filter	[output_filter = <STRING>]	output_filter = ‘ofilter.exe’	Optional	no output filter
Analysis driver	{analysis_driver = <STRING>}	analysis_driver = ‘analysis.exe’	Required	N/A
Failure capturing	[{failure_capture} {abort} | {retry = <INTEGER>} | {recover = <LISTof><REAL>} | {continuation}]	failure_capture retry = 5	Optional group	abort

Note that the recent additions of `concurrent_drivers`, `pre_driver`, and `post_driver` to `dakota.input.spec` is a placeholder for the level of parallelism involving concurrent analyses within a function evaluation (see Pending Extensions on page 104). This capability will be described in the V1.2 release where it will likely be merged with the existing facility to become `input_filter`, `analysis_drivers`, and `output_filter` (in which the use of a single analysis driver becomes a special case of the generalized specification).

In addition to the general application interface specifications, the type of application interface involves a selection between `system` or `direct` required group specifications. For system call application interfaces, `asynchronous`, `parameters_file`, `results_file`, `analysis_usage`, `aprepro`, `file_tag`, and `file_save` are additional settings within the group specification. Asynchronous function evaluations (system calls placed in the background with “&”) can be specified with the `asynchronous` specification, whereas the default behavior is synchronous function evaluations (system calls in the foreground). Refer to Enabling Software Components on page 100 for additional information on asynchronous procedures. The parameters and results file names are supplied as strings using the `parameters_file` and `results_file` specifications. Both specifications are optional with the default data transfer files being temporary files (e.g., `/usr/tmp/aaaa08861`, see Unix temporary files on page 83). The parameters and results file names are passed on the command line of the system calls (refer to 1-piece Interface on page 82 and 3-piece Interface on page 81 for details). The format of the data in these files is as described in DAKOTA File Data Formats on page 85 with the APREPRO format option for parameters files invoked via the `aprepro` specification. Special analysis command syntax can be entered as a string using `analysis_usage`. This special syntax replaces the `analysis_driver` portion of DAKOTA’s system call; however, it does not affect the `input_filter` and `output_filter` syntax (if filters are present). Its default is no special syntax (string default is `'DEFAULT'`), such that the `analysis_driver` will be used in the standard way as described in The System Call Application Interface on page 81. File tagging (appending parameters and results files with the function evaluation number; see File tagging on page 82) and file saving (leaving parameters and results files in existence after their use is complete; see File saving on page 82) are controlled

with the `file_tag` and `file_save` flags. If these specifications are omitted, the default is no file tagging (no appended function evaluation number) and no file saving (files will be removed after a function evaluation). File tagging is most useful when multiple function evaluations are running simultaneously using files in a shared disk space, and file saving is most useful when debugging the data communication between DAKOTA and the simulation. The additional specifications for system call application interfaces are summarized in Table 11. Refer to The System Call Application Interface on page 81 for additional details and examples.

Table 11 Additional specifications for system call application interfaces

Description	Specification	Sample	Status	Default
Application interface type	({system} ...)	system	Required group	N/A
Evaluation synchronization	[asynchronous]	asynchronous	Optional	synchronous evaluations
Parameters file name	[parameters_file = <STRING>]	parameters_file = ‘params.in’	Optional	Unix temp files
Results file name	[results_file = <STRING>]	results_file = ‘results.out’	Optional	Unix temp files
Special analysis usage syntax	[analysis_usage = <STRING>]	analysis_usage = ‘analysis.exe < params.in > results.out’	Optional	standard analysis usage
Aprepro format	[aprepro]	aprepro	Optional	standard format
File tag	[file_tag]	file_tag	Optional	no tagging
File save	[file_save]	file_save	Optional	no saving

For direct application interfaces, `asynchronous`, `evaluation_servers`, and `processors_per_evaluation` are additional settings within the required group. Asynchronous function evaluations (POSIX multithreading) can be specified with the `asynchronous` specification, whereas the default behavior is synchronous function evaluations (direct procedure calls). Refer to Enabling Software Components on page 100 for additional information on asynchronous procedures. The `evaluation_servers` and `processors_per_evaluation` specifications are used to configure multiprocessor partitions for multilevel parallelism. Typically, one or the other is specified to define how the processors allocated to an iterator are divided into multiprocessor evaluation servers; however, if both are specified and they are not in agreement, then `evaluation_servers` takes precedence. Refer to Specifying Parallelism on page 107 for additional details and examples on multiprocessor partitions. The direct application interface specifications are summarized in Table 12.

Table 12 Additional specifications for direct application interfaces

Description	Specification	Sample	Status	Default
Application interface type	({direct} ...)	direct	Required group	N/A
Evaluation synchronization	[asynchronous]	asynchronous	Optional	synchronous evaluations
Number of evaluation servers	[evaluation_servers = <INTEGER>]	evaluation_servers = 5	Optional	number of processors minus 1
Number of processors per evaluation	[processors_per_evaluation = <INTEGER>]	processors_per_evaluation = 256	Optional	1

Approximation Interface

The approximation interface uses an approximate representation of a true model (a surrogate model) to perform the parameter to response mapping. This approximation is built and updated using data from the true model as described in The Approximation Interface on page 95. Approximation interfaces are used extensively in the sequential approximate optimization strategy (see Sequential Approximate Optimization on page 74), in which the goal is to reduce expense by minimizing the number of function evaluations performed with the true model.

The approximation interface specification requires the specification of one of the following approximation methods: `neural_network`, `response_surface`, `multi_point`, or `mars_surface`. These specifications invoke a layered perceptron artificial neural network approximation (see the The ANN Approximation Interface on page 98), a quadratic polynomial response surface approximation (see The RSM Approximation Interface on page 97), a multipoint approximation (not yet available), or a multivariate adaptive regression spline approximation (see The MARS Approximation Interface on page 97), respectively. Table 13 summarizes the approximation interface specification.

Table 13 Specification detail for approximation interfaces

Description	Specification	Sample	Status	Default
Approximation interface	({approximation} ...)	approximation	Required group	N/A
Type	{neural_network} | {response_surface} | {multi_point} | {mars_surface}	neural_network	Required	N/A

Test Interface

The test interface uses an internally available test problem to perform the parameter to response mapping. These problems are compiled directly into the DAKOTA executable as part of the direct function application interface class (see The Direct Function Application Interface on page 80) and are used for algorithm testing.

The test interface specification requires the specification of a string to identify the test problem to be used. Table 14 summarizes this specification.

Table 14 Specification detail for test interfaces

Description	Specification	Sample	Status	Default
Test interface	{test = <STRING>}	test = ‘text_book’	Required	N/A

Currently, only the ‘`text_book`’ simulator is available as an internal test problem. Information on this problem is available in the Example Problems on page 328.

Variables Commands

Uncertain Variables on page 138

Description

The variables section in a DAKOTA input file specifies the parameter set to be iterated by a particular method. This parameter set is made up of design, uncertain, and state variable specifications. Design variables can be continuous or discrete and consist of those variables which an optimizer adjusts in order to locate an optimal design. Each of the design parameters can have an initial point, a lower bound, an upper bound, and a descriptive tag. Uncertain variables are continuous variables which are characterized by probability distributions. Each uncertain variable specification can contain a distribution type, a mean, a standard deviation, a lower bound, an upper bound, a histogram file name, and a descriptive tag. State variables can be continuous or discrete and consist of "other" variables which are to be mapped through the simulation interface. Each state variable specification can have an initial state and a descriptor. State variables provide a convenient mechanism for parameterizing additional model inputs, such as mesh density, simulation convergence tolerances and time step controls, and will be used to enact model adaptivity in future strategy developments.

Several examples follow. In the first example, two continuous design variables are specified:

```
variables,                                \
  continuous_design = 2                   \
    cdv_initial_point    0.9    1.1  \
    cdv_upper_bounds     5.8    2.9  \
    cdv_lower_bounds     0.5   -2.9  \
    cdv_descriptor   'radius' 'location'
```

In the next example, defaults are employed. In this case, `cdv_initial_point` will default to a vector of 0.0 values, `cdv_upper_bounds` will default to vector values of `DBL_MAX` (defined in the `float.h` C header file), `cdv_lower_bounds` will default to a vector of `-DBL_MAX` values, and `cdv_descriptor` will default to a vector of 'cdv_*i*' strings, where *i* goes from one to two:

```
variables,                                \
  continuous_design = 2
```

In the last example, a variables specification containing continuous and discrete design variables, uncertain variables, continuous and discrete state variables, and a set identifier is shown:

```
variables,\
  id_variables = 'V1'\
  continuous_design = 2\
    cdv_initial_point    0.9    1.1\
    cdv_upper_bounds     5.8    2.9       \
    cdv_lower_bounds     0.5   -2.9       \
    cdv_descriptor   'radius' 'location'\
  discrete_design = 1\
    ddv_initial_point    2\
    ddv_upper_bounds     1       \
```

```
    ddv_lower_bounds     3       \
    ddv_descriptor   'material'\
  uncertain = 2\
    uv_distribution_type = 'normal', 'lognormal' \
    uv_means = 250.0  480.0 \
    uv_std_deviations = 12.4  27.1\
    uv_lower_bounds = 220.0  410.0 \
    uv_upper_bounds = 280.0  550.0 \
    uv_descriptor = 'T_fail_1' 'T_fail_2'\
  continuous_state = 2\
    csv_initial_state = 1.e-4  1.e-6\
    csv_descriptor = 'EPSIT1'  'EPSIT2'\
  discrete_state = 1\
    dsv_initial_state = 100\
    dsv_descriptor = 'load_case'
```

The most general case of having a mixture of each of the different types of variables is supported within all of the iterators even though certain iterators will only modify certain types of variables (e.g., optimizers only modify design variables). This implies that variables which are not under the direct control of a particular iterator will be mapped through the interface unmodified for all evaluations of the iterator. This allows for a variety of parameterizations within the model in addition to those which are being used by a particular iterator.

Supporting the most general case is more difficult since decisions have to be made about how to appropriately size gradient vectors and Hessian matrices. Derivatives are never needed with respect to any discrete variables (since these derivatives do not exist) and the types of continuous variables for which derivatives are needed depend on the type of study being performed. For optimization and least squares problems, function derivatives are only needed with respect to the *continuous design variables* since this is the information used by the optimizer in computing a search direction. Similarly, for nondeterministic analysis methods which use gradient and/or Hessian information, function derivatives are only needed with respect to the *uncertain variables*. And lastly, parameter study methods which are cataloguing gradient and/or Hessian information do not draw a distinction among continuous variables; therefore, function derivatives must be supplied with respect to *all continuous variables* that are specified (continuous design, uncertain, and continuous state variables).

Specification

The variables specification has the following structure:

```
variables, \
  <set identifier>\
  <continuous design variables specification>\
  <discrete design variables specification>\
  <uncertain variables specification>\
  <continuous state variables specification>\
  <discrete state variables specification>
```

Referring to the IDR Input Specification File on page 112, it is evident from the enclosing brackets that the set identifier specification and the continuous design, discrete design, uncertain, continuous state, and discrete state variables specifications are all optional. The set identifier is a stand-alone optional specification, whereas the latter five are optional *group* specifications, meaning that the group can either appear or not as a unit. If any part of an optional group is specified, then all required parts of the group must appear.

The optional set identifier can be used to provide a unique identifier string for labeling a particular variables specification. A method can then identify the use of a particular set of variables by specifying this label in its `variables_pointer` specification (see Method Commands on page 156). The optional status of the continuous and discrete design, uncertain, and continuous and discrete state variables specifications allows the user to specify only those variables which are present (rather than explicitly specifying that the number of a particular type of variables = 0). However, at least one type of variables must have nonzero size or an input error message will result. The following sections describe each of these specification components in additional detail.

Set Identifier

The optional set identifier specification uses the keyword `id_variables` to input a string for use in identifying a particular variables set with a particular method (see also `variables_pointer` in the Method Commands on page 156). For example, a method whose specification contains `variables_pointer = 'V1'` will use a variables set with `id_variables = 'V1'`.

If the set identifier specification is omitted, a particular variables set will be used by a method only if that method omits specifying a `variables_pointer` and if the variables set was the last set parsed (or is the only set parsed). In common practice, if only one variables set exists, then `id_variables` can be safely omitted from the variables specification and `variables_pointer` can be omitted from the method specification(s), since there is no potential for ambiguity in this case. Table 15 summarizes the set identifier inputs.

Table 15 Specification detail for set identifier

Description	Specification	Sample	Status	Default
Variables set identifier	[id_variables = <STRING>]	id_variables = 'V1'	Optional	use of last variables parsed

Design Variables

Within the optional continuous design variables specification group, the number of continuous design variables is a required specification and the initial guess, lower bounds, upper bounds, and

variable names of the continuous design variables are optional specifications. Likewise, within the optional discrete design variables specification group, the number of discrete design variables is a required specification and the initial guess, lower bounds, upper bounds, and variable names of the discrete design variables are optional specifications. Default values for optional specifications include zeros for initial values, positive and negative machine limits for upper and lower bounds, and numbered strings for descriptors. Table 16 summarizes the details of the continuous design variable specification and Table 17 summarizes the details of the discrete design variable specification.

Table 16 Specification detail for continuous design variables

Description	Specification	Sample	Status	Default
Continuous design variables	[{continuous_design = <INTEGER>} ...]	continuous_design = 4	Optional group	no continuous design variables
Initial point	[cdv_initial_point = <LISTof><REAL>]	cdv_initial_point = 1.,2.1,0.3,4.2	Optional	Vector values = 0.0
Lower bounds	[cdv_lower_bounds = <LISTof> <REAL>]	cdv_lower_bounds = -1.,-2.,0.,-4.2	Optional	Vector values = `-DBL_MAX`
Upper bounds	[cdv_upper_bounds = <LISTof> <REAL>]	cdv_upper_bounds = 5.2,6.3,6.6,9.1	Optional	Vector values = `+DBL_MAX`
Descriptors	[cdv_descriptor = <LISTof> <STRING>]	cdv_descriptor = ‘c1’, ’c2’, ’c3’, ’c4’	Optional	Vector of ‘cdv_*i*’ where i = 1,2,3...

Table 17 Specification detail for discrete design variables

Description	Specification	Sample	Status	Default
Discrete design variables	[{discrete_design = <INTEGER>} ...]	discrete_design = 2	Optional group	no discrete design variables
Initial point	[ddv_initial_point = <LISTof> <INTEGER>]	ddv_initial_point = 3, 5	Optional	Vector values = 0
Lower bounds	[ddv_lower_bounds = <LISTof> <INTEGER>]	ddv_lower_bounds = 0, 0	Optional	Vector values = INT_MIN
Upper bounds	[ddv_upper_bounds = <LISTof> <INTEGER>]	ddv_upper_bounds = 10, 10	Optional	Vector values = INT_MAX
Descriptors	[ddv_descriptor = <LISTof> <STRING>]	ddv_descriptor = ‘d1’, ’d2’	Optional	Vector of ‘ddv_*i*’ where i = 1,2,3,...

The `cdv_initial_point` and `ddv_initial_point` specifications provide the point in design space from which an iterator is started for the continuous and discrete design variables, respectively. The `cdv_lower_bounds`, `ddv_lower_bounds`, `cdv_upper_bounds` and

ddv_upper_bounds restrict the size of the feasible design space and are frequently used to prevent nonphysical designs. The defaults for these bounds are linked to architecture constants (DBL_MAX, INT_MAX, INT_MIN) which are defined in the float.h and limits.h system header files. The cdv_descriptor and ddv_descriptor specifications supply strings which will be replicated through the Dakota output to help identify the numerical values for these parameters.

Uncertain Variables

Within the optional uncertain variables specification group, the number of uncertain variables and the distribution types are required specifications and the means, standard deviations, lower bounds, upper bounds, histogram file names, and descriptors are optional specifications. That is, if the uncertain variables group specification is included, then the number of uncertain variables and distribution types must be supplied at a minimum, whereas the other specifications in the group can rely on default values. Table 18 summarizes the details of the uncertain variable specification.

Table 18 Specification detail for uncertain variables specification

Description	Specification	Sample	Status	Default
Uncertain variables	[{uncertain = <INTEGER>} ...]	uncertain = 2	Optional group	no uncertain variables
Distribution type	{uv_distribution_type = <LISTof> <STRING>}	uv_distribution_type = ‘normal’, ‘lognormal’	Required	N/A
Means	[uv_means = <LISTof> <REAL>]	uv_means = 250., 480.	Optional	Vector values = 0
Standard deviations	[uv_std_deviations = <LISTof> <REAL>]	uv_std_deviations = 12.4, 27.1	Optional	Vector values = 0
Lower bounds	[uv_lower_bounds = <LISTof> <REAL>]	uv_lower_bounds = 220., 410.	Optional	Vector values = -DBL_MAX
Upper bounds	[uv_upper_bounds = <LISTof> <REAL>]	uv_upper_bounds = 280., 550.	Optional	Vector values = + DBL_MAX
Histogram file names	[uv_filenames = <LISTof> <STRING>]	uv_filenames = ‘T_fail1.dat’, ‘T_fail2.dat’	Optional	no histogram file names
Descriptors	[uv_descriptor = <LISTof> <STRING>]	uv_descriptor = ‘T_fail1’, ‘T_fail2’	Optional	Vector of ‘uv_*i*’ where i = 1,2,3,...

The uv_distribution_type vector identifies the type of distribution used to describe the statistics of each uncertain variable. Allowable distribution types are currently ‘normal’, ‘lognormal’, ‘constant’, ‘uniform’, ‘loguniform’, ‘weibull’, ‘logweibull’, and ‘histogram’. The

uv_means and uv_std_deviations specifications provide this data for those distributions which are characterized by means and standard deviations (normal and weibull are; constant, uniform, and histogram are not). Likewise, the uv_lower_bounds and uv_upper_bounds restrict the tails of the distributions for those distributions for which bounds are meaningful. Default bounds are linked to an architecture constant (DBL_MAX) defined in the float.h system header file. The uv_filenames specification provides the file names for variables of the histogram distribution type. The uv_descriptor specification provides strings which will be replicated through the Dakota output to help identify the numerical values for these parameters.

Each of the vector inputs, if specified, must be of length equal to the number of uncertain variables. Since certain distribution types may not have values for each of the uv_means, uv_std_deviations, uv_lower_bounds, uv_upper_bounds, and uv_filenames specifications, these arrays should be padded with place holders. For example, if uv_distribution_type = ‘normal’, ‘uniform’, ‘histogram’, then uv_std_deviations might equal 12.0, 0, 0 where the trailing 0’s are place holders in the array since uniform and histogram distributions do not specify standard deviations. Likewise, uv_filenames would be specified as ‘’, ‘’, ‘file.dat’ since only the histogram distribution type requires a file name specification. This behavior was chosen since it is believed to be more readable at a glance.

State Variables

Within the optional continuous state variables specification group, the number of continuous state variables and their initial states are required specifications and the continuous descriptor vector is an optional specification. Likewise, within the discrete state variables specification group, the number of discrete state variables and their initial states are required specifications and the discrete descriptor vector is an optional specification. These variables provide a convenient mechanism for managing additional model parameterizations such as mesh density, simulation convergence tolerances, and time step controls. Table 19 summarizes the details of the continuous state variable specification and Table 20 summarizes the details of the discrete state variable specification.

Table 19 Specification detail for continuous state variables

Description	Specification	Sample	Status	Default
Continuous state variables	[{continuous_state = <INTEGER>} ...]	continuous_state = 2	Optional group	No continuous state variables
Initial states	{csv_initial_state = <LISTof><REAL>}	csv_initial_state = 3.1, 4.2	Required	N/A
Descriptors	[csv_descriptor = <LISTof><STRING>]	csv_descriptor = ‘EPSIT1’, ‘EPSIT2’	Optional	Vector of ‘csv_*i*’ where *i* = 1,2,3,...

Table 20 Specification detail for discrete state variables

Description	Specification	Sample	Status	Default
Discrete state variables	[{discrete_state = <INTEGER>} ...]	discrete_state = 2	Optional group	No discrete state variables
Initial states	{dsv_initial_state = <LISTof><REAL>}	dsv_initial_state = 3, 4	Required	N/A
Descriptors	[dsv_descriptor = <LISTof><STRING>]	dsv_descriptor = ‘material1’, ‘material2’	Optional	Vector of ‘dsv_*i*’ where *i* = 1,2,3,...

The `csv_initial_state` and `dsv_initial_state` specifications define the initial values for the continuous and discrete state variables which will be passed through to the simulator (e.g., in order to define parameterized modeling controls). The `csv_descriptor` and `dsv_descriptor` vectors provide strings which will be replicated through the Dakota output to help identify the numerical values for these parameters.

Responses Commands

Active Set Vector Diagram on page 143

Description

The responses specification in a DAKOTA input file specifies the data set that can be recovered from the interface during the course of iteration. This data set is made up of a set of functions, their first derivative vectors (gradients), and their second derivative matrices (Hessians). This abstraction provides a generic data container (the **DakotaResponse** class) whose contents are interpreted differently depending upon the type of iteration being performed. In the case of optimization, the set of functions consists of an objective function (or objective functions in the case of multiobjective optimization) and nonlinear constraints. Linear constraints are not part of a response set since their coefficients can be communicated to an optimizer at startup and then computed internally for all function evaluations (see NPSOL Method on page 162). In the case of least squares iterators, the functions consist of individual residual terms (**not** the sum of the squares objective function; this function is computed internally by the least squares iterators). In the case of nondeterministic iterators, the function set is made up of generic response functions for which the effect of parameter uncertainty is to be quantified. Lastly, parameter study iterators may be used with any of the response data set types. Within the C++ implementation, the same data structures are used to provide each of these response data set types; only the interpretation of the data varies from iterator branch to iterator branch.

Gradient availability may be described by `no_gradients`, `numerical_gradients`, `analytic_gradients`, or `mixed_gradients`. “`no_gradients`” means that gradient information is not needed in the study. “`numerical_gradients`” means that gradient information is needed and will be computed with finite differences using either the native or one of the vendor finite differencing routines. “`analytic_gradients`” means that gradient information is available directly from the simulation (finite differencing is not required). And “`mixed_gradients`” means that some gradient information is available directly from the simulation whereas the rest will have to be finite differenced.

Hessian availability may be described by `no_hessians` or `analytic_hessians` where the meanings are the same as for the corresponding gradient availability settings. Numerical Hessians are not currently supported, since, in the case of optimization, this would imply a finite difference-Newton technique for which a direct algorithm already exists. Capability for numerical Hessians can be added if the need arises.

The responses specification provides a description of the data set that is available for use by the iterator *during the course of its iteration*. This should be distinguished from the data set described in an active set vector (see DAKOTA File Data Formats on page 85) which describes the subset of the available data needed *on a particular function evaluation*. Put another way, the responses specification is a broad description of the data that is available whereas the active set vector describes the particular subset of the available data that is currently needed.

Several examples follow. The first example shows an optimization data set of an objective function and two nonlinear constraints. These three functions have analytic gradient availability and no Hessian availability.

```
responses,\
   num_objective_functions = 1\
   num_nonlinear_constraints = 2\
   analytic_gradients\
   no_hessians
```

The next example shows a specification for a least squares data set. The six residual functions will have numerical gradients computed using the dakota finite differencing routine with central differences of 0.1% (plus/minus delta value = .001*value).

```
responses,\
   num_least_squares_terms = 6\
   numerical_gradients\
   method_source dakota\
   interval_type central\
   fd_step_size = .001\
   no_hessians
```

The last example shows a specification that could be used with a nondeterministic iterator. The three response functions have no gradient or Hessian availability; therefore, only function values will be used by the iterator.

```
responses,\
   num_response_functions = 3\
   no_gradients\
   no_hessians
```

Parameter study iterators are not restricted in terms of the response data sets which may be catalogued; they may be used with any of the function specification examples shown above.

Specification

The responses specification has the following structure:

```
responses, \
   <set identifier>\
   <active set vector usage>\
   <function specification>\
   <gradient specification>\
   <hessian specification>
```

Referring to the IDR Input Specification File on page 112, it is evident from the enclosing brackets that the set identifier and the active set vector usage specifications are optional. However, the function, gradient, and Hessian specifications are all required specifications, each of which contains several possible specifications separated by logical OR’s. The function specification must be one of three types: 1) a group containing objective and constraint functions, 2) a least squares terms specification, or 3) a response functions specification. The gradient specification must be one of four types: 1) no gradients, 2) numerical gradients, 3) analytic

gradients, or 4) mixed gradients. And the Hessian specification must be either 1) no Hessians or 2) analytic Hessians.

The optional set identifier can be used to provide a unique identifier string for labeling a particular responses specification. A method can then identify the use of a particular response set by specifying this label in its `responses_pointer` specification (see Method Commands on page 156). The active set vector usage setting allows the user to turn off active set distinctions (default is on) so that a simulation interface can neglect to include active set logic (at the possible penalty of wasted computations). The function, gradient, and Hessian specifications define the data set that can be recovered from the interface. The following sections describe each of these specification components in additional detail.

Set Identifier

The optional set identifier specification uses the keyword `id_responses` to input a string for use in identifying a particular responses set with a particular method (see also `responses_pointer` in the Method Commands on page 156). For example, a method whose specification contains `responses_pointer = 'R1'` will use a responses set with `id_responses = 'R1'`.

If this specification is omitted, a particular responses set will be used by a method only if that method omits specifying a `responses_pointer` and if the responses set was the last set parsed (or is the only set parsed). In common practice, if only one responses set exists, then `id_responses` can be safely omitted from the responses specification and `responses_pointer` can be omitted from the method specification(s), since there is no potential for ambiguity in this case. Table 21 summarizes the set identifier inputs.

Table 21 Specification detail for set identifier

Description	Specification	Sample	Status	Default
Responses set identifier	[id_responses = <STRING>]	id_responses = 'R1'	Optional	use of last responses parsed

Active Set Vector Usage

A future capability will be the option to turn the active set vector (ASV) usage `on` or `off`. Currently, only the default `on` setting is supported; its behavior is described in DAKOTA File Data Formats on page 85. Setting the ASV control to `off` will cause Dakota to always request a "full" data set (the full function, gradient, and Hessian data that is available in the problem as specified in the responses specification) ***on each function evaluation***. For example, if ASV control is `off` and the responses section specifies four response functions, analytic gradients, and no Hessians, then the ASV on *every* function evaluation will be a vector of length four

containing all threes, regardless of what subset of this data is currently needed. While wasteful of computations in many instances, this removes the need for ASV-related logic in user-built interfaces. That is, ASV control set to `on` will result in requests of only that specific data which is needed on each evaluation and will require the user's interface to read the ASV requests and perform the appropriate logic in conditionally returning only the data requested. Conversely, ASV control set to `off` removes the need for this additional logic and allows the user to return the same data set on every evaluation. In general, the default `on` behavior is recommended for efficiency through the elimination of unnecessary computations, although in some instances, ASV control set to `off` can simplify operations and speed filter development for time critical applications.

Note that in all cases, the data returned to DAKOTA from the user's interface must match the ASV passed in (or else a response recovery error will result). The important observation is that when ASV control is `off`, the ASV vector values do not change from one evaluation to the next. Therefore their content need not be checked on every evaluation. Table 22 summarizes the active set vector usage setting.

Table 22 Specification detail for active set vector usage specification

Description	Specification	Sample	Status	Default
Active set vector usage	[{active_set_vector} {on} | {off}]	active_set_vector on	Optional group	on

Function specification

The function specification must be one of three types: 1) a group containing objective and constraint functions, 2) a least squares terms specification, or 3) a response functions specification. These function sets correspond to optimization, least squares, and uncertainty quantification iterators, respectively. Parameter study iterators may be used with any of the three function specifications.

Objective and Constraint Functions (Optimization Data Set)

An optimization data set is specified using `num_objective_functions`, and optionally `num_nonlinear_constraints`. Multiobjective optimization is not yet supported within the optimizer branch, so `num_objective_functions` should be set to one when using DOT, NPSOL, OPT++, or SGOPT. Direct input of linear constraints can be used to improve the efficiency of NPSOL (see `linear_constraints` in Method Independent Controls on page 158). However, DOT, OPT++, and SGOPT do not yet support specialized handling of linear constraints; in these cases, any linear constraints should be included in the more general `num_nonlinear_constraints` count. Table 23 summarizes the optimization data set specification.

Table 23 Specification detail for optimization data sets

Description	Specification	Sample	Status	Default
Number of objective functions	({num_objective_functions = <INTEGER>} ...)	num_objective_functions = 1	Required group	N/A
Number of nonlinear constraints	[num_nonlinear_constraints = <INTEGER>]	num_nonlinear_constraints = 2	Optional	0

Least Squares Terms (Least Squares Data Set)

A least squares data set is specified using `num_least_squares_terms`. Each of these terms is a residual function to be driven towards zero. These types of problems are commonly encountered in parameter estimation and model validation. Least squares problems are most efficiently solved using special-purpose least squares solvers such as Gauss-Newton or Levenberg-Marquardt; however, they may also be solved using general-purpose optimization algorithms. It is important to realize that, while DAKOTA can solve these problems with either least squares or optimization algorithms, the response data sets to be returned from the simulator are different. Least squares involves a set of residual functions whereas optimization involves a single objective function (sum of the squares of the residuals). Therefore, derivative data in the least squares case involves derivatives of the least squares terms, whereas the optimization case involves derivatives of the sum of the squares objective function. Switching between the two approaches will likely require different simulation interfaces capable of returning the different granularity of response data required. Table 24 summarizes the least squares data set specification.

Table 24 Specification detail for nonlinear least squares data sets

Description	Specification	Sample	Status	Default
Number of Least Squares Terms	{num_least_squares_terms = <INTEGER>}	num_least_squares_terms = 20	Required	N/A

Response Functions (Generic Data Set)

A generic response data set is specified using `num_response_functions`. Each of these functions is simply a response quantity of interest with no special interpretation taken by the method in use. This type of data set is used by uncertainty quantification methods, in which the effect of parameter uncertainty on response functions is quantified, and can also be used in parameter studies (although parameter studies are not restricted to this data set), in which the effect of parameter variations on response functions is evaluated. Whereas objective, constraint, and residual functions have special meanings within the data sets used by optimization and least squares algorithms (i.e., their usage is linked to their identity), the response functions in an

uncertainty quantification or parameter study need not have a specific interpretation. This is due primarily to the fact that the values of these response functions are not fed back to these algorithms as a basis for additional iterative improvement. Therefore, the user is free to define whatever functional form is convenient. Table 25 summarizes the generic response function data set specification.

Table 25 Specification detail for generic response function data sets

Description	Specification	Sample	Status	Default
Number of Response Functions	{num_response_functions = <INTEGER>}	num_response_functions = 2	Required	N/A

Gradient specification

The gradient specification must be one of four types: 1) no gradients, 2) numerical gradients, 3) analytic gradients, or 4) mixed gradients.

No Gradients

The `no_gradients` specification means that gradient information is not needed in the study. Therefore, it will neither be retrieved from the simulation nor computed with finite differences. `no_gradients` is a complete specification for this case.

Numerical Gradients

The `numerical_gradients` specification means that gradient information is needed and will be computed with finite differences using either the native or one of the vendor finite differencing routines. The `method_source` setting specifies the source of the finite difference routine that will be used to compute the numerical gradients: `dakota` denotes DAKOTA's internal finite differencing algorithm and `vendor` denotes the finite differencing algorithm supplied by the iterator package in use (DOT, NPSOL, and OPT++ each have their own internal finite differencing routines). The `vendor` routine was chosen as the default since certain libraries modify their algorithm when they are aware that finite differencing is being performed. Since the `dakota` routine hides this fact from the optimizers (the optimizers are configured to accept user-supplied gradients, which they assume to be of analytic accuracy), the potential exists for the `vendor` setting to trigger the use of an algorithm more optimized for the higher expense and/or lower accuracy of finite-differencing (e.g., NPSOL uses gradients in its line search when in user-supplied gradient mode, but uses a value-based line search procedure when internally finite differencing). However, while this algorithm modification may reduce expense in serial operations, the `dakota` routine is preferable when seeking to exploit the parallelism in finite difference evaluations (see Exploiting Parallelism on page 99). And in fact, NPSOL's use

of gradients in its line search (user-supplied gradient mode) provides excellent load balancing for parallel optimization without need to resort to speculative optimization approaches. The `interval_type` setting is used to select between `forward` and `central` differences in the numerical gradient calculations. The DAKOTA, DOT, and OPT++ routines have both `forward` and `central` differences available, and NPSOL starts with `forward` differences and automatically switches to `central` differences as the iteration progresses (the user has no control over this). Lastly, `fd_step_size` specifies the ***relative*** finite difference step size to be used in the computations. For DAKOTA, DOT, and OPT++, the intervals are computed by multiplying the `fd_step_size` with the current parameter value. In this case, a minimum absolute differencing interval is needed when the current parameter value is close to zero. This prevents finite difference intervals for the parameter which are too small to distinguish differences in the response quantities being computed. DAKOTA, DOT, and OPT++ all use `1.e-2*fd_step_size` as their minimum absolute differencing interval. With a `fd_step_size = .001`, for example, DAKOTA, DOT, and OPT++ will use intervals of `.001*`current value with a minimum interval of `1.e-5`. NPSOL uses a different formula for its finite difference intervals: `fd_step_size*(1+|`current parameter value`|)`. This definition has the advantage of eliminating the need for a minimum absolute differencing interval since the interval no longer goes to zero as the current parameter value goes to zero. Table 26 summarizes the numerical gradient specification.

Table 26 Specification detail for numerical gradients

Description	Specification	Sample	Status	Default
Numerical gradients	({numerical_gradients} ...)	numerical_gradients	Required group	N/A
Method source	[{method_source} {dakota} | {vendor}]	method_source, dakota	Optional group	vendor
Interval Type	[{interval_type} {forward} | {central}]	interval_type, forward	Optional group	forward
Finite difference step size	[fd_step_size = <REAL>]	fd_step_size = 0.001	Optional	0.001

Analytic Gradients

The `analytic_gradients` specification means that gradient information is available directly from the simulation (finite differencing is not required). The simulation must return the gradient data in the DAKOTA format (see DAKOTA File Data Formats on page 85) for the case of file transfer of data. `analytic_gradients` is a complete specification for this case.

Mixed Gradients

The `mixed_gradients` specification means that some gradient information is available directly from the simulation (analytic) whereas the rest will have to be finite differenced

(numerical). This specification is useful since it is generally wise to make use of as much analytic gradient information as is available and then to finite difference for the rest. For example, the objective function may be a simple analytic function of the design variables (e.g., weight) whereas the constraints are nonlinear implicit functions of complex analyses (e.g., maximum stress). The `id_analytic` list specifies by number the functions which have analytic gradients, and the `id_numerical` list specifies by number the functions which must use numerical gradients. The `method_source`, `interval_type`, and `fd_step_size` specifications are as described previously under the Numerical Gradients on page 146 specification and pertain to those functions listed by the `id_numerical` list. Table 27 summarizes the mixed gradient specification.

Table 27 Specification detail for mixed gradients

Description	Specification	Sample	Status	Default
Mixed gradients	({mixed_gradients} ...)	mixed_gradients	Required group	N/A
Analytic derivatives function list	{id_analytic = <LISTof> <INTEGER>	id_analytic = 2,4	Required	N/A
Numerical derivatives function list	{id_numerical = <LISTof> <INTEGER>}	id_numerical = 1,3,5	Required	N/A
Method source	[{method_source} {dakota} | {vendor}]	method_source, dakota	Optional group	vendor
Interval Type	[{interval_type} {forward} | {central}]	interval_type, forward	Optional group	forward
Finite difference step size	[fd_step_size = <REAL>]	fd_step_size = 0.001	Optional	0.001

Hessian specification

Hessian availability must be specified with either `no_hessians` or `analytic_hessians`. Numerical Hessians are not currently supported, since, in the case of optimization, this would imply a finite difference-Newton technique for which a direct algorithm already exists. Capability for numerical Hessians can be added if the need arises.

No Hessians

The `no_hessians` specification means that the method does not require Hessian information. Therefore, it will neither be retrieved from the simulation nor computed with finite differences. `no_hessians` is a complete specification for this case.

Analytic Hessians

The `analytic_hessians` specification means that Hessian information is available directly from the simulation. The simulation must return the Hessian data in the DAKOTA format (see DAKOTA File Data Formats on page 85) for the case of file transfer of data. `analytic_hessians` is a complete specification for this case.

Strategy Commands

Description

The strategy section in a DAKOTA input file specifies the top level technique which will govern the management of iterators and models in the solution of the problem of interest. Five strategies currently exist: `multi_level`, `seq_approximate_opt`, `opt_under_uncertainty`, `branch_and_bound`, and `single_method`. In a `multi_level` optimization strategy, a list of methods is specified which will be used synergistically in seeking an optimal design.The goal here is to exploit the strengths of different optimization algorithms through different stages of the optimization process. Global/local hybrids (e.g., genetic algorithms combined with nonlinear programming) are a common example in which the desire for a global optimum is balanced with the need for efficient navigation to a local optimum. In sequential approximate optimization (`seq_approximate_opt`), a set of points is selected from a design and analysis of computer experiments (DACE) and evaluated with the simulation model. These results are then used to build an approximate model, such as a response surface or an artificial neural network. An optimizer iterates on this approximate model and computes an approximate optimum. This point is evaluated with the simulation model and the measured improvement in the simulation model is used to modify the boundaries (i.e., trust region) of the approximation. The approximation is then updated with the new point and additional approximate optimization cycles are performed until convergence. The goal with `seq_approximate_opt` is to reduce the total number of simulations required for the optimization. In optimization under uncertainty (`opt_under_uncertainty`), a nondeterministic iterator is used to evaluate the effect of uncertain variable distributions on responses of interest. These responses and/or their statistics are then included in the objective and constraint functions of an optimization process. The nondeterministic iteration may be nested within the optimization iteration, nested with approximations, or segregated in an uncoupled approach. In the branch and bound strategy (`branch_and_bound`), mixed continuous/discrete applications can be solved through parameter domain decomposition and relaxation of integrality constraints. Lastly, the `single_method` strategy provides the means for simple execution of a single iterator.

The specification for `multi_level` involves a list of method identifier strings, and each of the corresponding method specifications (see Method Commands on page 156) has the responsibility for identifying the variables, interface, and responses specifications that each method will use. The `seq_approximate_opt` strategy must specify one iterator, an approximate interface, and an actual interface. The same variables and responses specifications will be used by both interfaces. The `opt_under_uncertainty` strategy must specify the optimization and nondeterministic iterators and, again, each of the corresponding method specifications points to the variables, interface, and responses specifications to be used (which, in this case, will likely be different since optimization and nondeterministic methods use different data sets). The `branch_and_bound` strategy must specify one iterator and the number of concurrent iterator servers to be utilized. The `single_method` strategy may specify a method identifier which in

turn specifies the variables, interface, and responses identifiers, or it may specify nothing additional and invoke the default behavior of employing the last specifications parsed. Invoking the default behavior is particularly appropriate if only one specification is present for method, variables, interface, and responses since there is no source for confusion in this case. In addition, `single_method` is the default strategy if no strategy specification is supplied.

Example specifications for the five strategies follow. A `multi_level` example is:

```
strategy, \
  multi_level uncoupled\
  method_list = 'GA1', 'CPS1', 'NLP1'
```

A `seq_approximate_opt` example specification is:

```
strategy, \
  seq_approximate_opt\
  opt_method = 'NLP1'\
  approximate_interface = 'resp_surf'\
  actual_interface = 'simulation'
```

An `opt_under_uncertainty` example specification is:

```
strategy, \
  opt_under_uncertainty\
  opt_method = 'NLP1'\
  nond_method = 'LHS_MC'
```

A `branch_and_bound` example specification is:

```
strategy, \
  branch_and_bound\
  opt_method = 'NLP1'\
  iterator_servers = 4
```

A `single_method` example specification is:

```
strategy, \
  single_method\
  method_pointer = 'NLP1'
```

In addition to management of multiple iterators and models, the strategy layer manages the division of operations between master and slave processors. Refer to Exploiting Parallelism on page 99 for additional details.

Specification

The strategy specification has the following structure:

```
strategy, \
  <single_method> or <multi_level> or <seq_approximate_opt>
    or
    <opt_under_uncertainty> or <branch_and_bound>
```

Referring to the IDR Input Specification File on page 112, it is evident that the five strategy specifications (`multi_level`, `seq_approximate_opt`, `opt_under_uncertainty`,

branch_and_bound, or single_method) are required groups (enclosing in parentheses) separated by OR's. Thus, one and only one strategy specification must be provided.

The various strategy specifications identify the methods and models (or more specifically, interfaces) that will be employed in the strategy as well as controls for interaction (e.g., switching) between the methods and models. The methods and models are specified using string pointers that correspond to identifier strings in the method and interface specifications (such as 'method1' or 'interface1'). They should NOT be confused with method selections (such as dot_mmfd) or interface types (such as application). The following sections describe each of these strategy specifications in additional detail.

Single Method Commands

The single_method strategy may be specified using the single_method keyword by itself, or an optional method_pointer may additionally be used to point to a particular method. For example, method_pointer = 'NLP1' points to the method whose specification contains id_method = 'NLP1'. If method_pointer is not used, then the last method specification parsed will be used as the iterator. Invoking this default behavior is most appropriate if only one method specification is present since there is no potential source of confusion in this case. Table 28 summarizes the single_method strategy inputs.

Table 28 Specification detail for single_method strategies

Description	Specification	Sample	Status	Default
Single method strategy	({single_method} ...)	single_method	Required group	N/A
Method pointer	[method_pointer = <STRING>]	method_pointer = 'NLP1'	Optional	use of last method parsed

Refer to Single Method on page 71 for an overview of the single_method objects and algorithm logic.

Multilevel Hybrid Optimization Commands

The multi_level hybrid optimization strategy has uncoupled, uncoupled adaptive, and coupled approaches (see Multilevel Hybrid Optimization on page 71 for more information on the algorithms employed). In the two uncoupled approaches, a list of method strings supplied with the method_list specification specifies the identity and sequence of iterators to be used. Any number of iterators may be specified. The uncoupled adaptive approach may be specified by turning on the adaptive flag. If the adaptive flag in specified, then progress_threshold must also be specified since it is a required part of the optional group

specification. In the nonadaptive case, method switching is managed through the separate convergence controls of each method. In the adaptive case, however, method switching occurs when the internal progress metric (normalized between 0.0 and 1.0) falls below the user specified `progress_threshold`. Table 29 summarizes the uncoupled `multi_level` strategy inputs.

Table 29 Specification detail for uncoupled multi_level strategies

Description	Specification	Sample	Status	Default
Multi-level strategy	({multi_level} ...)	multi_level	Required group	N/A
uncoupled approach	({uncoupled} ...)	uncoupled	Required group	N/A
adaptive control	[{adaptive} {progress_threshold = <REAL>}]	adaptive, progress_threshold = 0.5	Optional group	no adaptive control
List of methods	{method_list = <LISTof> <STRING}	method_list = ‘GA1’, ‘CPS1’, ‘NLP1’	Required	N/A

In the coupled approach, global and local method strings supplied with the `global_method` and `local_method` specifications identify the two methods to be used. The `local_search_probability` setting is as optional specification for supplying the probability (between 0.0 and 1.0) of employing local search to improve estimates within the global search. Table 30 summarizes the coupled `multi_level` strategy inputs.

Table 30 Specification detail for coupled multi_level strategies

Description	Specification	Sample	Status	Default
Multi-level strategy	({multi_level} ...)	multi_level	Required group	N/A
coupled approach	({coupled} ...)	coupled	Required group	N/A
Global method	{global_method = <STRING>}	global_method = ‘GA1’	Required	N/A
Local method	{local_method = <STRING>}	local_method = ‘NLP1’	Required	N/A
Local search probability	[local_search_probability = <REAL>]	local_search_probability = 0.5	Optional	0.1

In either the uncoupled or coupled case, each of the methods listed is responsible for cross-referencing its own variables, interface, and responses specifications (using `interface_pointer`, `variables_pointer`, and `responses_pointer`; see Method Independent Controls on page 158) within its method specification.

Sequential Approximate Optimization Commands

The `seq_approximate_opt` strategy must specify an iterator using `opt_method`, an approximate interface using `approximate_interface`, and an application interface using `actual_interface`. The method specification identified by `opt_method` is responsible for pointing to the variables and responses specifications that will be used by both interfaces (using `variables_pointer` and `responses_pointer`; see Method Independent Controls on page 158). Table 31 summarizes the `seq_approximate_opt` strategy inputs.

Table 31 Specification detail for seq_approximate_opt strategies

Description	Specification	Sample	Status	Default
Sequential approximate optimization strategy	({seq_approximate_opt} ...)	seq_approximate_opt	Required group	N/A
Optimization method	{opt_method = <STRING>}	opt_method = ‘NLP1’	Required	N/A
Approximate interface	{approximate_interface = <STRING>}	approximate_interface = ‘resp_surf’	Required	N/A
Actual interface	{actual_interface = <STRING>}	actual_interface = ‘simulation’	Required	N/A

Refer to Sequential Approximate Optimization on page 74 for an overview of the `seq_approximate_opt` objects and algorithm logic.

Optimization Under Uncertainty Commands

The `opt_under_uncertainty` strategy must specify an optimization iterator using `opt_method` and a nondeterministic iterator using `nond_method`. The method specifications identified by `opt_method` and `nond_method` are responsible for pointing to the variables, interface, and responses specifications to be used by these methods (using `interface_pointer`, `variables_pointer`, and `responses_pointer`; see Method Independent Controls on page 158). Since optimization and nondeterministic iteration use very different types of data, the variables, interface, and responses specifications used by these methods will often be distinct. Table 32 summarizes the `opt_under_uncertainty` strategy inputs.

Table 32 Specification detail for opt_under_uncertainty strategies

Description	Specification	Sample	Status	Default
Optimization under uncertainty strategy	({opt_under_uncertainty} ...)	opt_under_uncertainty	Required group	N/A

Table 32 Specification detail for opt_under_uncertainty strategies

Description	Specification	Sample	Status	Default
Optimization method	{opt_method = <STRING>}	opt_method = ‘NLP1’	Required	N/A
Nondeterministic method	{nond_method = <STRING>}	nond_method = ‘LHS_MC’	Required	N/A

Refer to Optimization Under Uncertainty on page 75 for an overview of the `opt_under_uncertainty` objects and algorithm logic.

Branch and Bound Commands

The `branch_and_bound` strategy must specify an iterator using `opt_method` and the number of concurrent iterator servers using `iterator_servers`. The method specification identified by `opt_method` is responsible for pointing to the variables, interface, and responses specifications that will be used by the method (using `interface_pointer`, `variables_pointer`, and `responses_pointer`; see Method Independent Controls on page 158). Table 33 summarizes the `branch_and_bound` strategy inputs.

Table 33 Specification detail for branch_and_bound strategies

Description	Specification	Sample	Status	Default
Branch and bound strategy	({branch_and_bound} ...)	branch_and_bound	Required group	N/A
Optimization method	{opt_method = <STRING>}	opt_method = ‘NLP1’	Required	N/A
Concurrent iterator servers	{iterator_servers = <INTEGER>}	iterator_servers = 4	Required	N/A

Refer to Branch and Bound on page 76 for an overview of the `branch_and_bound` objects and algorithm logic.

Method Commands

Description

The method section in a DAKOTA input file specifies the name and controls of an iterator. The terms “method” and “iterator” can be used interchangeably, although method usually refers to an input specification whereas iterator usually refers to an object within the **DakotaIterator** hierarchy. A method specification, then, is used to select an iterator from the iterator hierarchy (see Iterator and Strategy Hierarchies on page 52), which includes optimization, uncertainty quantification, least squares, and parameter study iterators (see Capability Overview on page 59 for more information on these iterator branches). This iterator may be used alone or with other iterators as dictated by the strategy specification (refer to Strategy Commands on page 150 for strategy command syntax and to Strategy Capabilities on page 70 for strategy algorithm descriptions).

Several examples follow. The first example shows a specification for an optimization method.

```
method,\
  dot_sqp\
    id_method = ‘NLP1’\
    variables_pointer = ‘V1’\
    interface_pointer = ‘I1’\
    responses_pointer = ‘R1’
```

This example demonstrates the use of identifiers and pointers. The method specification has been tagged with the string `‘NLP1’`. This string can be used in a strategy specification to identify that this method will be invoked by the strategy. Similarly, variables, interface, and responses specifications which have been tagged elsewhere with `‘V1’`, `‘I1’`, and `‘R1’` strings are being identified as the specifications that this method will use in its iteration. Note that this type of tagging and cross-referencing is not needed when relationships among specifications are unambiguous (due to the presence of only one specification). The next example shows a specification for a least squares method.

```
method,\
  optpp_g_newton\
    convergence_tolerance = 1.e-8\
    max_iterations = 10\
    search_method, trust_region\
    gradient_tolerance = 1.e-6
```

This example demonstrates some method independent and method dependent controls. The `convergence_tolerance` and `max_iterations` settings are method independent controls, in that they are defined for a variety of methods. The `search_method` and `gradient_tolerance` settings are method dependent controls, in that they are only meaningful for OPT++ methods. The next example shows a specification for a nondeterministic iterator.

```
method,\
```

```
  nond_probability\
    observations = 100\
    seed = 1\
    sample_type, lhs\
    response_thresholds = 1000., 500.
```

Each of the nondeterministic method controls are method dependent controls. The last example shows a specification for a parameter study iterator where, again, each of the controls are method dependent.

```
method,\
  parameter_study\
    step_vector = 1.,1.,1.\
    num_steps = 10
```

Specification

The method specification has the following structure:

```
method, \
  <method independent controls>\
  <method selection>\
  <method dependent controls>
```

where `<method selection>` is one of the following:

```
dot_frcg, dot_mmfd, dot_bfgs, dot_slp, dot_sqp, npsol_sqp,
optpp_cg, optpp_q_newton, optpp_g_newton, optpp_newton,
optpp_fd_newton, optpp_baq_newton, optpp_ba_newton,
optpp_bcq_newton, optpp_bcg_newton, optpp_bc_newton,
optpp_bc_ellipsoid, optpp_pds, optpp_test_new,
sgopt_pga_real, sgopt_pga_int, sgopt_coord_ps,
sgopt_coord_sps, sgopt_solis_wets, sgopt_strat_mc,
nond_probability, nond_mean_value, vector_parameter_study,
list_parameter_study, centered_parameter_study,
multidim_parameter_study
```

The `<method independent controls>` are those controls which are valid for a variety of methods. In some cases, these controls are abstractions which may have slightly different implementations from one method to the next. The `<method dependent controls>` are those controls which are only meaningful for a specific method or library. Referring to the IDR Input Specification File on page 112, the `<method independent controls>` are those controls defined externally from and prior to the method selection blocks. They are all optional. The method selection blocks are all required group specifications separated by logical OR's. The `<method dependent controls>` are those controls defined within the method selection blocks. The following sections provide additional detail on the method independent controls followed by the method selections and their corresponding method dependent controls.

Method Independent Controls

The method independent controls include a method identifier string, pointers to variables, interface, and responses specifications, speculative gradient selection, output verbosity control, linear constraint specification, convergence tolerance specification, and maximum iteration and function evaluation limits. While each of these controls is not valid for every method, the controls are valid for enough methods that it was reasonable to pull them out of the method dependent blocks and consolidate the specifications.

The method identifier string is supplied with `id_method` and is used to provide a unique identifier string for use with strategy specifications. It is appropriate to omit a method identifier string if only one method is included in the input file and `single_method` is the selected strategy, since the binding of a strategy to a method is unambiguous in this case.

The `interface_pointer`, `variables_pointer`, and `responses_pointer` specifications in the method keyword provide strings for cross-referencing with the `id_interface`, `id_variables`, and `id_responses` string inputs from the interface, variables, and responses keyword specifications. These pointers identify which specifications will be used by a particular method for its mapping of variables into responses through an interface. If a pointer string is specified and no corresponding id is available, an error message will be printed. If no pointer string is specified, the last specification parsed will be used. It is appropriate to omit this cross-referencing whenever the relationships are unambiguous due to the presence of only one specification. Since the method specification is responsible for cross-referencing with the interface, variables, and responses specifications, identification of methods at the strategy layer is often sufficient to completely specify all of the object interrelationships.

When performing gradient-based optimization in parallel, `speculative` gradients can be selected to address the load imbalance that can occur between gradient evaluation and line search phases. In a typical synchronous analysis, the line search phase consists primarily of evaluating the objective function and any constraints at a trial point, and then testing the trial point for a sufficient decrease in the objective function value and/or constraint violation. If a sufficient decrease is not observed, then one or more additional trial points may be attempted in series. However, if the trial point is accepted then the line search phase is complete and the gradient evaluation phase begins. By speculating that the gradient information associated with a given line search trial point will be used later, additional coarse grained parallelism can be introduced during an asynchronous analysis. This is achieved by computing the gradient information, either by finite difference or analytically, in parallel, at the same time as the line search phase trial-point function values. This balances the total amount of computation to be performed at each design point and allows for efficient utilization of multiple processors. While the total amount of work performed will generally increase (since some speculative gradients will not be used when a trial point is rejected in the line search phase), the run time will usually decrease (since gradient evaluations needed at the start of each new optimization cycle were already performed in parallel during the line search phase). Refer to [Byrd, R.H., Schnabel, R.B., and Schultz, G.A., 1988] for additional details. The `speculative` specification is implemented for the gradient-based

optimizers in the DOT, NPSOL, and OPT++ libraries, and it can be used with `dakota numerical` or `analytic` gradient selections in the responses specification (see Gradient specification on page 146). It should not be selected with `vendor numerical` gradients since vendor internal finite difference algorithms have not been modified for this purpose. In full-Newton approaches, the Hessian is also computed speculatively.

Output verbosity control is specified with `output` followed by either `verbose` or `quiet`. This control is mapped into each iterator to manage the volume of data that is returned to the user during the course of the iteration. Different iterators implement this control in slightly different ways, however the meaning is consistent.

Linear constraint coefficients can be supplied with the `linear_constraints` list specification. While many of DAKOTA's optimizers will eventually support specialized handling of linear constraints, currently only the NPSOL library supports this feature. For all other optimizers, linear constraints should be included within the more general `num_nonlinear_constraints` count and returned on every function evaluation. For NPSOL, linear constraints need not be computed by the user's interface on every function evaluation; rather the coefficients of the linear constraints can be provided to NPSOL at startup, allowing NPSOL to track the linear constraints internally. Note that linear constraints are those constraints that are linear in the *design variables*, e.g.:

$$3x_1 - 4x_2 \leq 0.5$$
$$x_1 + x_2 \geq 2.0$$

which is not to be confused with something like

$$\sigma(X) - \sigma_{fail} \leq 0$$

which is linear in a response quantity, but the response quantity is a nonlinear implicit function of the design variables. For the linear constraints above, the specification would appear as:

```
linear_constraints = 3.0, -4.0, -0.5, -1.0, -1.0, 2.0
```

where the list in divided into individual constraints based on the number of continuous design variables and according to the following assumed form (which was selected for consistency with the nonlinear constraint assumed form of $g_i(X) \leq 0$):

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_0 \leq 0$$

The `convergence_tolerance` specification provides a real value for controlling the termination of iteration. In most cases, it is a relative convergence tolerance for the objective function; i.e., if the change in the objective function between successive iterations divided by the previous objective function is less than the amount specified by `convergence_tolerance`, then this convergence criterion is satisfied on the current iteration. Since no progress may be made on one iteration followed by significant progress on a subsequent iteration, some libraries require that the `convergence_tolerance` be satisfied on two or more consecutive iterations prior to termination of iteration. This control is most meaningful for optimization and least squares iterators and is not currently implemented within the uncertainty quantification and parameter study iterator branches. Refer to the DOT, NPSOL, OPT++, and SGOPT specifications for the specific interpretation of `convergence_tolerance` for these libraries.

The `constraint_tolerance` specification determines the maximum allowable value of infeasibility that any constraint in an optimization problem may possess at termination. It is specified as a positive real value. If a constraint function is greater than this value then it is considered to be violated by the optimization algorithm. This specification gives some control over how tightly the constraints may be satisfied at convergence of the algorithm. However, if the value is set too small the algorithm may terminate with one or more constraints being violated. This specification is currently meaningful for the NPSOL and DOT constrained optimizers.

The `max_iterations` and `max_function_evaluations` controls provide integer limits for the maximum number of iterations and maximum number of function evaluations, respectively. The difference between an iteration and a function evaluation is that a function evaluation involves a single parameter to response mapping through an interface, whereas an iteration involves a complete cycle of computation within the iterator. Thus, an iteration generally involves multiple function evaluations (e.g., for descent direction and line search computations in gradient-based optimization, population and multiple offset evaluations in nongradient-based optimization, etc.). This control is not currently implemented within the uncertainty quantification and parameter study iterator branches, and in the case of optimization and least squares, does not currently capture function evaluations that occur as part of the `method_source dakota` finite difference routine (since these additional evaluations are intentionally isolated from the iterators). Table 34 provides the specification detail for the method independent controls.

Table 34 Specification detail for the method independent controls

Description	Specification	Sample	Status	Default
Method set identifier	[id_method = <STRING>]	id_method = ‘NLP1’	Optional	strategy usage of last method parsed
Interface pointer	[interface_pointer = <STRING>]	interface_pointer = ‘I1’	Optional	method usage of last interface parsed
Variables pointer	[variables_pointer = <STRING>]	variables_pointer = ‘V1’	Optional	method usage of last variables parsed
Responses pointer	[responses_pointer = <STRING>]	responses_pointer = ‘R1’	Optional	method usage of last responses parsed
Speculative gradients and Hessians	[speculative]	speculative	Optional	standard gradients and Hessians
Output verbosity	[{output} {verbose} | {quiet}]	output verbose	Optional group	quiet
Linear constraints	[linear_constraints = <LISTof> <REAL>]	linear_constraints = 1.0, 2.0, 3.0	Optional	no linear constraints

Table 34 Specification detail for the method independent controls

Description	Specification	Sample	Status	Default
Constraint tolerance	[constraint_ tolerance = <REAL>	constraint_ tolerance = 1.e-4	Optional	Optimization code dependent
Convergence tolerance	[convergence_ tolerance = <REAL>]	convergence_ tolerance = 1.e-5	Optional	1.e-4
Maximum iterations	[max_iterations = <INTEGER>]	max_iterations = 10	Optional	100
Maximum function evaluations	[max_function_ evaluations = <INTEGER>]	max_function_ evaluations =200	Optional	1000

Developer's notes: defaults for method independent and method dependent controls are defined in Dakota/src/DataMethod.C.

DOT Methods

The DOT library [Vanderplaats Research and Development, 1995] contains nonlinear programming optimizers, specifically the Broyden-Fletcher-Goldfarb-Shanno (Dakota's `dot_bfgs` method) and Fletcher-Reeves conjugate gradient (Dakota's `dot_frcg` method) methods for unconstrained optimization, and the modified method of feasible directions (Dakota's `dot_mmfd` method), sequential linear programming (Dakota's `dot_slp` method), and sequential quadratic programming (Dakota's `dot_sqp` method) methods for constrained optimization. DAKOTA implements the DOT library within the **DOTOptimizer** class.

Method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during a DOT optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. This convergence criterion must be satisfied for two consecutive iterations before DOT will terminate. The `constraint_tolerance` specification defines how tightly constraint functions are to be satisfied at convergence. The default value for DOT constrained optimizers is `0.003`. Extremely small values for `constraint_tolerance` may not be attainable. The `output` verbosity specification controls the amount of information generated by DOT: the `quiet` setting results in header information, final results, and objective function, constraint, and parameter information on each iteration; whereas the `verbose` setting adds additional information on gradients, search direction, one-dimensional search results, and parameter scaling factors. DOT contains no parallel algorithms which can directly take advantage of asynchronous evaluations. However, if

numerical_gradients with method_source dakota is specified, then an asynchronous interface specification will trigger the use of asynchronous evaluations for the finite difference function evaluations. In addition, if speculative is specified, then gradients (dakota numerical or analytic gradients) will be computed on each line search evaluation in order to balance the load and lower the total run time in parallel optimization studies. Lastly, specialized handling of linear_constraints is not supported with DOT; linear constraints should be included within the num_nonlinear_constraints count and returned on every function evaluation. Specification detail for these method independent controls is provided in Table 34.

Developer's notes: max_iterations, max_function_evaluations, convergence_tolerance, and output verbosity are implemented within **DOTOptimizer** as follows: max_iterations is mapped into DOT's ITMAX parameter within its IPRM array; max_function_evaluations is implemented directly in the **DOTOptimizer::find_optimum** loop since there is no DOT parameter equivalent; convergence_tolerance is mapped into DOT's DELOBJ parameter (the relative convergence tolerance) within its RPRM array; and output verbosity is mapped into DOT's IPRINT parameter within its function call parameter list (verbose: IPRINT = 7; quiet: IPRINT = 3). Refer to [Vanderplaats Research and Development, 1995] for information on IPRM, RPRM, and the DOT function call parameter list.

Method dependent controls

DOT's only method dependent control is optimization_type which may be either minimize or maximize. DOT has the only methods within DAKOTA which provide this control; to convert a maximization problem into the minimization formulation assumed by other methods, simply change the sign on the objective function (i.e., multiply by -1). Table 35 provides the specification detail for the DOT methods and their method dependent controls.

Table 35 Specification detail for the DOT methods

Description	Specification	Sample	Status	Default
DOT method	({dot_bfgs} ...) | ({dot_frcg} ...) | ({dot_mmfd} ...) | ({dot_slp} ...) | ({dot_sqp} ...)	dot_sqp	Required group	N/A
Optimization type	[{optimization_type} {minimize} | {maximize}]	optimization_type, minimize	Optional group	minimize

Developer's notes: optimization_type is mapped into DOT's MINMAX parameter within its function call parameter list.

NPSOL Method

The NPSOL library [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] contains a sequential quadratic programming (SQP) implementation (the npsol_sqp method). SQP is a

nonlinear programming optimizer for constrained minimization. DAKOTA implements the NPSOL library within the **NPSOLOptimizer** class.

Method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major SQP iterations and the number of function evaluations that can be performed during an NPSOL optimization. The `convergence_tolerance` control defines NPSOL's internal optimality tolerance which is used in evaluating if an iterate satisfies the first-order Kuhn-Tucker conditions for a minimum. The magnitude of `convergence_tolerance` approximately specifies the number of significant digits of accuracy desired in the final objective function (e.g., `convergence_tolerance = 1.e-6` will result in approximately six digits of accuracy in the final objective function). The `constraint_tolerance` control defines how tightly the constraint functions are satisfied at convergence. The default value is dependent upon the machine precision of the platform in use, but is typically on the order of `1.e-8` for double precision computations. Extremely small values for `constraint_tolerance` may not be attainable. The `output` verbosity setting controls the amount of information generated at each major SQP iteration: the `quiet` setting results in only one line of diagnostic output for each major iteration and prints the final optimization solution, whereas the `verbose` setting adds additional information on the objective function, constraints, and variables at each major iteration.

NPSOL is not a parallel algorithm and cannot directly take advantage of asynchronous evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, an `asynchronous` interface specification will trigger the use of asynchronous evaluations for the finite difference function evaluations. An important related observation is the fact that NPSOL uses two different line searches depending on how gradients are computed. For either `analytic_gradients` or `numerical_gradients` with `method_source dakota`, NPSOL is placed in user-supplied gradient mode (NPSOL's "Derivative Level" is set to 3) and it uses a gradient-based line search (presumably since it assumes that the user-supplied gradients are inexpensive). On the other hand, if `numerical_gradients` are selected with `method_source vendor`, then NPSOL is computing finite differences internally and it will use a value-based line search (presumably since it assumes that finite differencing on each line search evaluation is too expensive). The ramifications of this are: (1) performance will vary between `method_source dakota` and `method_source vendor` for `numerical_gradients`, and (2) gradient speculation is unnecessary when performing optimization in parallel since the gradient-based line search in user-supplied gradient mode is already load balanced for multiple processor execution. Therefore, a `speculative` specification will be ignored by NPSOL, and optimization with numerical gradients should select `method_source dakota` for load balanced parallel operation and `method_source vendor` for efficient serial operation.

Lastly, NPSOL supports specialized handling of linear constraints with the `linear_constraints` list specification. By specifying the coefficients of the linear constraints, this information can be provided to NPSOL at initialization and tracked internally, removing the need for the user to provide the values of the linear constraints on every function evaluation. Refer to Method Independent Controls on page 158 for additional information and to Table 34 for method independent control specification detail.

Developer's notes: `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, and `output` verbosity are implemented within **NPSOLOptimizer** as follows: `max_iterations` is mapped into NPSOL's `"Major Iteration Limit"` parameter using its `NPOPTN` routine; `max_function_evaluations` is implemented directly in **NPSOLOptimizer**'s evaluator functions since there is no NPSOL parameter equivalent; `convergence_tolerance` is mapped into NPSOL's `"Optimality Tolerance"` parameter using the NPOPTN routine; `output` verbosity is mapped into NPSOL's `"Major Print Level"` parameter using the NPOPTN routine (`verbose`: `Major Print Level` = 20; `quiet`: `Major Print Level` = 10). Refer to [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] for information on NPSOL's optional input parameters and the NPOPTN subroutine.

Method dependent controls

NPSOL's method dependent controls are `verify_level`, `function_precision`, and `linesearch_tolerance`. The `verify_level` control instructs NPSOL to perform finite difference verifications on user-supplied gradient components. The `function_precision` control provides NPSOL an estimate of the accuracy to which the problem functions can be computed. This is used to prevent NPSOL from trying to distinguish between function values that differ by less than the inherent error in the calculation. And the `linesearch_tolerance` setting controls the accuracy of the line search. The smaller the value (between 0 and 1), the more accurately NPSOL will attempt to compute a precise minimum along the search direction. Table 36 provides the specification detail for the NPSOL SQP method and its method dependent controls.

Table 36 Specification detail for the NPSOL SQP method

Description	Specification	Sample	Status	Default
NPSOL's SQP method	({npsol_sqp} ...)	npsol_sqp	Required group	N/A
Verify level	[verify_level = <INTEGER>]	verify_level = -1	Optional	-1 (no gradient verification)
Function precision	[function_precision = <REAL>]	function_precision = 1.e-6	Optional	1.e-10
Line search tolerance	[linesearch_tolerance = <REAL>]	linesearch_tolerance = 0.4	Optional	0.9 (inaccurate line search)

Developer's notes: `verify_level`, `function_precision`, and `linesearch_tolerance` are mapped into NPSOL's `"Verify Level"`, `"Function Precision"` and `"Linesearch Tolerance"` parameters, respectively, using NPSOL's `NPOPTN` routine. Refer to [Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986] for additional information on these controls.

OPT++ Methods

The OPT++ library [Meza, J.C., 1994] contains primarily nonlinear programming optimizers for unconstrained minimization: Polak-Ribiere conjugate gradient (DAKOTA's `optpp_cg` method), quasi-Newton, barrier function quasi-Newton, and bound constrained quasi-Newton (DAKOTA's `optpp_q_newton`, `optpp_baq_newton`, and `optpp_bcq_newton` methods), Gauss-Newton and bound constrained Gauss-Newton (DAKOTA's `optpp_g_newton` and `optpp_bcg_newton` methods - part of DAKOTA's nonlinear least squares branch), full Newton, barrier function full Newton, and bound constrained full Newton (DAKOTA's `optpp_newton`, `optpp_ba_newton`, and `optpp_bc_newton` methods), finite difference Newton (DAKOTA's `optpp_fd_newton` method), and bound constrained ellipsoid (DAKOTA's `optpp_bc_ellipsoid` method). The library also contains a directed search algorithm, PDS (parallel direct search, DAKOTA's `optpp_pds` method), and an input place holder for new algorithm testing (DAKOTA's `optpp_test_new` method). DAKOTA implements the OPT++ library within the **SNLLOptimizer** class, where "SNLL" denotes Sandia National Laboratories - Livermore.

Method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during an OPT++ optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. The `output` verbosity specification controls the amount of information generated by OPT++: the `quiet` setting corresponds to turning OPT++'s internal debug mode off, whereas the `verbose` setting turns debug mode on. OPT++'s gradient-based methods are not parallel algorithms and cannot directly take advantage of asynchronous evaluations. However, if `numerical_gradients` with `method_source dakota` is specified, an `asynchronous` interface specification will trigger the use of asynchronous evaluations for the finite difference gradient computations. OPT++'s nongradient-based PDS method can directly exploit asynchronous evaluations; however, this capability has not been implemented within DAKOTA V1.1.

The `speculative` specification enables speculative computation of Hessian and/or gradient information, where applicable, for load balancing purposes. The specification is applicable to the computation of gradient information in cases where `trust_region` or `value_based_line_search` methods can be applied. See the OPT++ Method dependent controls on page 162 for a description of `value_based_line_search` and `trust_region` methods. The `speculative` specification must be used in conjunction with `dakota` `numerical` or `analytic` gradients. The specification is ignored and a warning message is printed for gradient computations when a `gradient_based_line_search` is used, or when the `optpp_ba_newton`, `optpp_baq_newton` or `optpp_bc_ellipsoid`

methods are used. The speculative specification can also be applied to the full Newton methods, which require computation of `analytic hessians,` or for the `optpp_fd_newton` method. However, the specification is ignored for the `optpp_g_newton` Hessian computation, which approximates the Hessian from function and gradient values.

Lastly, specialized handling of `linear_constraints` is not supported with OPT++; many OPT++ methods must be unconstrained and some can handle bound constraints. Specification detail for these method independent controls is provided in Table 34.

Developer's notes: within the **SNLLOptimizer** class, `max_iterations`, `max_function_evaluations`, and `convergence_tolerance` are set using OPT++'s `SetMaxIter`, `SetMaxFeval`, and `SetFcnTol` member functions, respectively; `output` verbosity is used to toggle OPT++'s debug mode using the `SetDebug` member function. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/VendorOptimizers/opt++ directory for information on OPT++ class member functions.

Method dependent controls

OPT++'s method dependent controls are `max_step`, `gradient_tolerance`, `search_method`, `initial_radius`, and `search_scheme_size`. The `max_step` control specifies the maximum step that can be taken when computing a change in the objective function iterate (e.g., limiting the Newton step computed from current gradient and Hessian information). It is equivalent to a move limit or a maximum trust region size. The `gradient_tolerance` control defines the threshold value on the L2 norm of the objective function gradient that indicates convergence to an unconstrained minimum (no active bound constraints). The `gradient_tolerance` control is defined for all gradient-based optimizers.

The `search_method` control is defined for all Newton-based optimizers and is used to select between `trust_region, gradient_based_line_search,` and `value_based_line_search` methods. The `max_step` control is applicable wherever one of the above `search_method` techniques is used. The `trust_region` search method is the default for all methods except ellipsoid, barrier, and bound-constrained methods. The ellipsoid and barrier methods use built-in directional searches, and thus, the overall `search_method` control does not apply. The use of trust region techniques for the bound-constrained methods is an open research issue, and currently the `line_search` method is the default. The `initial_radius` control is defined for the ellipsoid method to specify the initial radius of the ellipsoid, and `search_scheme_size` is defined for the PDS method to specify the number of points tospeculative gradient specification be used in the direct search template.

Table 37, Table 38, Table 39, Table 40, Table 41, and Table 42 provide the specification detail for the OPT++ methods and their method dependent controls. Table 37 covers the OPT++ conjugate gradient method specification. Table 38 provides the detail for all of the unconstrained and bound-constrained Newton-based methods. Table 39 provides the detail for barrier Newton methods. Table 40 provides the detail for the bound constrained ellipsoid method. Table 41 provides the detail for the parallel direct search method. And Table 42 provides the specification detail for OPT++ new method testing.

Table 37 Specification detail for the OPT++ conjugate gradient method

Description	Specification	Sample	Status	Default
OPT++'s conjugate gradient method	({optpp_cg} ...)	optpp_cg	Required group	N/A
Maximum step size	[max_step = <REAL>]	max_step = 1000.	Optional	1000.
Gradient tolerance	[gradient_tolerance = <REAL>]	gradient_tolerance = 0.0001	Optional	0.0001

Table 38 Specification detail for unconstrained and bound-constrained Newton-based OPT++ methods

Description	Specification	Sample	Status	Default
OPT++ Newton-based methods	({optpp_q_newton} ...) | ({optpp_g_newton} ...) | ({optpp_newton} ...) | ({optpp_fd_newton} ...) | ({optpp_bc_newton} ...) | ({optpp_bcq_newton} ...) | ({optpp_bcg_newton} ...)	optpp_q_newton	Required group	N/A
Search method	[{search_method} {value_based_line_search}| {gradient_based_line_searh}| {trust_region}]	search_method, value_based_line_search	Optional group	line_search for bc methods, trust_region for others
Maximum step size	[max_step = <REAL>]	max_step = 1000.0	Optional	1000.
Gradient tolerance	[gradient_tolerance = <REAL>]	gradient_tolerance = 0.0001	Optional	0.0001

Table 39 Specification detail for barrier-constrained Newton OPT++ methods

Description	Specification	Sample	Status	Default
OPT++ barrier Newton methods	({optpp_ba_newton} ...) | ({optpp_baq_newton} ...)	optpp_ba_newton	Required group	N/A
Gradient tolerance	[gradient_tolerance = <REAL>]	gradient_tolerance = 0.0001	Optional	0.0001

Table 40 Specification detail for the OPT++ bound constrained ellipsoid method

Description	Specification	Sample	Status	Default
OPT++'s bound constrained ellipsoid	({optpp_bc_ellipsoid} ...)	optpp_bc_ellipsoid	Required group	N/A
Initial radius	[initial_radius = <REAL>]	initial_radius = 1000.0	Optional	1000.
Maximum step size	[max_step = <REAL>]	max_step = 1000.	Optional	1000.
Gradient tolerance	[gradient_tolerance = <REAL>]	gradient_tolerance = 0.0001	Optional	0.0001

Table 41 Specification detail for the OPT++ PDS method

Description	Specification	Sample	Status	Default
OPT++'s Parallel Direct Search	({optpp_pds} ...)	optpp_pds	Required group	N/A
Search scheme size	[search_scheme_size = <INTEGER>]	search_scheme_size = 32	Optional	32

Table 42 Specification detail for OPT++ new method testing

Description	Specification	Sample	Status	Default
Placeholder for new OPT++ method testing	{optpp_test_new}	optpp_test_new	Required	N/A

Developer's notes: `max_step`, `gradient_tolerance`, `search_method`, `initial_radius`, and `search_scheme_size` are set using OPT++'s `SetMaxStep`, `SetGradTol`, `SetSearchStrategy`, `SetInitialEllipsoid`, and `SetSSS` member functions, respectively. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/VendorOptimizers/opt++ directory for information on OPT++ class member functions.

SGOPT Methods

The SGOPT (Stochastic Global OPTimization) library [Hart, W.E., 1997] contains a variety of global optimization algorithms, with an emphasis on stochastic methods. SGOPT currently includes the following global optimization methods: genetic algorithms (`sgopt_pga_real`, `sgopt_pga_int`) and stratified Monte Carlo (`sgopt_strat_mc`). Evolutionary pattern search algorithms, simulated annealing, tabu search, and multistart local search (to become part of DAKOTA's `coupled multi_level` strategy) are global methods which are under development but are not available in DAKOTA V1.0. Additionally, SGOPT includes several local

search algorithms such as Solis-Wets (`sgopt_solis_wets`) and deterministic and stochastic coordinate pattern search (`sgopt_coord_ps` and `sgopt_coord_sps`). DAKOTA implements the SGOPT library within the **SGOPTOptimizer** class.

Developer's notes: To specify method controls and options, DAKOTA's **SGOPTOptimizer** class instantiates SGOPT method interface objects (e.g., **IPGAreal** is an interface class to the **PGAreal** optimizer class). The purpose of these interface classes is to simplify the communication of information from driver programs (e.g., DAKOTA) to the SGOPT optimizer classes. This information transfer occurs through the passing of string data using the **process** member function available in the interface classes. For example, the command

```
baseOptimizerInterface->process("debug", "5");
```

uses a pointer to an optimizer interface object (`baseOptimizerInterface`) to set the `debug` data structure within the interface object's corresponding optimizer class to the integer 5.

Method independent controls

The method independent controls for `max_iterations` and `max_function_evaluations` limit the number of major iterations and the number of function evaluations that can be performed during an SGOPT optimization. The `convergence_tolerance` control defines the threshold value on relative change in the objective function that indicates convergence. The `output` verbosity specification controls the amount of information generated by SGOPT: the `quiet` setting corresponds to a low level of diagnostics reported only on those iterations for which improvement in the objective is observed, whereas the `verbose` setting corresponds to a higher level of diagnostics reported on every iteration. Many of SGOPT's nongradient-based methods have independent function evaluations that can directly take advantage of DAKOTA's parallel capabilities. The following methods currently support concurrent function evaluations: `sgopt_pga_real`, `sgopt_pga_int`, `sgopt_strat_mc`, `sgopt_coord_ps`, and `sgopt_coord_sps`. This methods automatically utilize asynchronous logic when utilizing multiple processors of when specifying an `asynchronous` interface. Note that parallel usage of `sgopt_coord_ps` or `sgopt_coord_sps` overrides any setting for `exploratory_moves` (see Coordinate pattern search (CPS) on page 172), since the `standard`, `offset`, `best_first`, and `biased_best_first` settings only involve relevant distinctions for the case of serial operation. Lastly, neither `speculative` gradients nor specialized handling of `linear_constraints` are supported with SGOPT since SGOPT methods are unconstrained and nongradient-based. Specification detail for method independent controls is provided in Table 34.

Developer's notes: `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, and `output` verbosity are implemented within **SGOPTOptimizer** as follows: `max_iterations` is mapped into SGOPT's `max_iters` data attribute using the **process** command available in SGOPT's interface classes; `max_function_evaluations` is mapped into `max_neval` using **process**; `convergence_tolerance` is mapped into `ftol` using **process**; `output` verbosity is mapped into `debug` and `dynamic_debug` settings using **process** (`verbose`: the `debug` level is set to 5 and the `dynamic_debug` flag is not set; `quiet`: the `debug` level is set to 0 and the `dynamic_debug` flag is turned on). The `dynamic_debug` flag determines whether results are reported on every iteration (off) or only on those iterations for which improvement in the objective is observed (on). SGOPT methods assume asynchronous operations whenever the algorithm has independent evaluations which can be performed simultaneously (implicit parallelism). Therefore, the `evaluations asynchronous` control is not mapped into the method (exception: `emcase` is set to 3 using **process** for asynchronous coordinate pattern searches),

rather it is used in **SGOPTRealApplication** and **SGOPTIntApplication** to control whether or not an asynchronous evaluation request from the method is honored by the model. Refer to [Hart, W.E., 1997] for additional information on SGOPT objects and controls.

Method dependent controls

`solution_accuracy` and `max_cpu_time` are method dependent controls which are defined for all SGOPT methods. Solution accuracy defines a convergence criterion in which the optimizer will terminate if it finds an objective function value lower than the specified accuracy. Note that the default of 1.e-5 should be overridden in those applications where it could cause premature termination. The maximum CPU time setting is another convergence criterion in which the optimizer will terminate if its CPU usage in seconds exceeds the specified limit. Table 43 provides the specification detail for these method dependent controls.

Table 43 Specification detail for SGOPT method dependent controls

Description	Specification	Sample	Status	Default
Solution Accuracy	[solution_accuracy = <REAL>]	solution_accuracy = 0.0	Optional	1.e-5
Maximum CPU Time	[max_cpu_time = <REAL>]	max_cpu_time = 86400.0	Optional	No limit

Developer's notes: `solution_accuracy` and `max_cpu_time` are passed into SGOPT's optimizers using **process** with identifiers of `acc` and `time`, respectively.

Each SGOPT method supplements the settings of Table 43 with controls which are specific to its particular class of method. Genetic algorithms have additional settings for random seed, population size, selection pressure, replacement, crossover, and mutation. Coordinate pattern search algorithms have additional settings for random seed (stochastic pattern search only), expansion policy, number of successes before expansion, expansion and contraction exponents, initial and threshold deltas, and exploratory moves selection. Solis-Wets has additional settings for random seed, number of successes before expansion, number of failures before contraction, and initial and threshold rho settings. And lastly, stratified Monte Carlo has additional settings for random seed and parameter space partitioning.

Genetic algorithms (GAs)

DAKOTA currently implements two types of GAs: a real-valued GA (`sgopt_pga_real`) and an integer-valued GA (`sgopt_pga_int`). Most controls for these two methods are the same, although their crossover and mutation controls have slight differences. Table 44 provides the specification detail for the controls which are common between the two GAs.

Table 44 Specification detail for the SGOPT GA methods

Description	Specification	Sample	Status	Default
GA methods	({sgopt_pga_real} ...) | ({sgopt_pga_int} ...)	sgopt_pga_real	Required group	N/A
Random seed	[seed = <INTEGER>]	seed = 1	Optional	1
population size	[population_size = <INTEGER>]	population_size = 10	Optional	100
selection pressure	[{selection_pressure} {rank = <REAL>} | {proportional}]	selection_pressure, rank = 2.0	Optional group	proportional
replacement type	[{replacement_type} {random} | {CHC} | {elitist} [new_solutions_generated = <INTEGER>]]	replacement_type elitist, new_solutions_generated = 5	Optional group	???

The random `seed` control provides a mechanism for making a stochastic optimization repeatable. For example, even though many of the processes within a genetic algorithm have random character, the use of the same random seed in identical studies will generate identical results. This, of course, implies that generating meaningful statistics on GA performance will require the user to vary the random seed on multiple runs. The `population_size` control specifies how many individuals will comprise the GA's population. The `selection_pressure` controls how strongly differences in fitness are weighted in the process of selecting "parents" for crossover. The `replacement_type` controls how current populations and newly generated individuals are combined into a new population.

Table 45 and Table 46 show the crossover and mutation controls which differ between `sgopt_pga_real` and `sgopt_pga_int`.

Table 45 Specification detail for SGOPT real GA crossover and mutation

Description	Specification	Sample	Status	Default
crossover type	[{crossover_type} {two_point} | {mid_point} | {blend} | {uniform} [crossover_rate = <REAL>]]	crossover_type mid_point, crossover_rate = 0.6	Optional group	two_point crossover with rate = 0.8
mutation type	[{mutation_type} ({normal} [std_deviation = <REAL>]) | {interval} | {cauchy} [dimension_rate = <REAL>] [population_rate = <REAL>]]	mutation_type normal, dimension_rate = 0.8	Optional group	???

Table 46 Specification detail for SGOPT integer GA crossover and mutation

Description	Specification	Sample	Status	Default
crossover type	[{crossover_type} {two_point} | {uniform} [crossover_rate = <REAL>]]	crossover_type uniform, crossover_rate = 0.6	Optional group	two_point crossover with rate = 0.8
mutation type	[{mutation_type} {offset} | {interval} [dimension_rate = <REAL>] [population_rate = <REAL>]]	mutation_type offset, dimension_rate = 0.8	Optional group	???

The `crossover_type` controls what approach is employed for combining parent genetic information to create offspring, and the `crossover_rate` specifies the probability of a crossover operation being performed to generate a new offspring. The `mutation_type` controls what approach is employed in randomly modifying design variables within the GA population. The associated `population_rate` controls the probability of mutation being performed on a particular individual, and if it is to be performed on an individual, the `dimension_rate` is used to govern the probability of mutation per design variable for the individual.

Coordinate pattern search (CPS)

DAKOTA implements two types of CPS: a deterministic CPS (`sgopt_coord_ps`) and a stochastic CPS (`sgopt_coord_sps`). Their controls are identical except that the stochastic CPS specification contains a random seed whereas the deterministic CPS specification does not. Table 47 provides the specification detail for SGOPT CPS methods and their method dependent controls.

Table 47 Specification detail for the SGOPT CPS methods

Description	Specification	Sample	Status	Default
CPS methods	({sgopt_coord_ps} ...) | ({sgopt_coord_sps} ...)	sgopt_coord_ps	Required group	N/A
Random seed (stochastic only)	[seed = <INTEGER>]	seed = 1	Optional	1
expansion policy	[{expansion_policy} {unlimited} | {once}]	expansion_policy once	Optional group	unlimited
expand after success	[expand_after_success = <INTEGER>]	expand_after_success = 2	Optional	1
expansion exponent	[expansion_exponent = <INTEGER>]	expansion_exponent = 1	Optional	0

Table 47 Specification detail for the SGOPT CPS methods

Description	Specification	Sample	Status	Default
contraction exponent	[contraction_exponent = <INTEGER>]	contraction_exponent = 1	Optional	-1
initial delta	{initial_delta = <REAL>}	initial_delta = 1.0	Required	N/A
threshold delta	{threshold_delta = <REAL>}	threshold_delta = 1.e-6	Required	N/A
exploratory moves	[{exploratory_moves} {standard} | {offset} | {best_first} | {biased_best_first}]	exploratory_moves best_first	Optional group	standard

As described previously, the random `seed` is used to make stochastic optimizations repeatable. The `expansion_policy` setting specifies how many times an increase in delta can occur (either `once` or `unlimited` times). The `expand_after_success` control specifies how many successful objective function improvements must occur with a specific delta prior to expansion of the delta. The `expansion_exponent` and `contraction_exponent` specify the exponents used to evaluate the expansion and contraction factors, respectively. The `initial_delta` and `threshold_delta` specify the starting delta value and the minimum value of delta that will be used prior to terminating, respectively. Lastly, the `exploratory_moves` setting controls how:

- the evaluations about a current point are ordered. The `offset` case examines each of the `2n` offsets in order whereas the `standard`, `best_first`, and `biased_best_first` examine each of the `n` dimensions in order. The offset and dimension orderings are identical in the deterministic case; the distinction is only relevant for stochastic CPS in which the orderings are shuffled either by offset or dimension (the order of the `n` dimensions is shuffled in the `best_first` and `biased_best_first` cases, and the order of the `2n` evaluations is shuffled in the `offset` case).
- whether or not the algorithm immediately selects the first improving point found (`offset`, `best_first`, and `biased_best_first`) or waits and selects the best improving point found from all new design points (`standard` as well as the parallel case).
- whether the algorithm uses a bias to guide the algorithm in a direction where improving points have previously been found (`biased_best_first`).

It is important to emphasize that the same sets of evaluation points are used by the `sgopt_coord_ps` and `sgopt_coord_sps` methods; it is only the *ordering* of the evaluations that can differ due to the shuffling in the stochastic case. Consequently, in the parallel case where the ordering of the evaluations is unimportant (since they are being performed simultaneously), `sgopt_coord_ps` and `sgopt_coord_sps` are essentially identical.

Solis-Wets

DAKOTA's implementation of SGOPT also contains the Solis-Wets algorithm. Table 48 provides the specification detail for this method and its method dependent controls.

Table 48 Specification detail for the SGOPT Solis-Wets method

Description	Specification	Sample	Status	Default
Solis-Wets method	({sgopt_solis_wets} ...)	sgopt_solis_wets	Required group	N/A
Random seed	[seed = <INTEGER>]	seed = 1	Optional	1
expand after success	[expand_after_success = <INTEGER>]	expand_after_success = 2	Optional	5
contract after failure	[contract_after_failure = <INTEGER>]	contract_after_failure = 2	Optional	3
initial ρ	[initial_rho = <REAL>]	initial_rho = 1.0	Optional	0.5
threshold ρ	[threshold_rho = <REAL>]	threshold_rho = 1.e-6	Optional	0.00001

As for other SGOPT methods, the random `seed` is used to make stochastic optimizations repeatable. Similar to CPS, `expand_after_success` specifies how many successful cycles must occur with a specific ρ prior to expansion of ρ. And `contract_after_failure` specifies how many unsuccessful cycles must occur with a specific ρ prior to contraction of ρ. The `initial_rho` and `threshold_rho` settings specify the starting ρ value and the minimum value of ρ that will be used prior to terminating, respectively.

Stratified Monte Carlo

Lastly, DAKOTA's implementation of SGOPT contains a stratified Monte Carlo (sMC) algorithm. Table 49 provides the specification detail for this method and its method dependent controls.

Table 49 Specification detail for the SGOPT sMC method

Description	Specification	Sample	Status	Default
sMC method	({sgopt_strat_mc} ...)	sgopt_strat_mc	Required group	N/A
Random seed	[seed = <INTEGER>]	seed = 1	Optional	1
partitions	[partitions = <LISTof> <INTEGER>]	partitions = 2, 4, 3	Optional	No partitioning

As for other SGOPT methods, the random `seed` is used to make stochastic optimizations repeatable. And the `partitions` list is used to specify the number of partitions in each design

variable. For example, `partitions = 2, 4, 3` specifies 2 partitions in the first design variable, 4 partitions in the second design variable, and 3 partitions in the third design variable.

Nondeterministic Methods

DAKOTA's nondeterministic branch does not currently make use of the method independent controls for `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, `speculative` gradients, `output` verbosity, or `linear_constraints`. As such, the nondeterministic branch documentation which follows is limited to the method dependent controls for the Monte Carlo probability and mean value methods.

Monte Carlo Probability Method

The Monte Carlo probability iterator is selected using the `nond_probability` specification. This iterator performs sampling for different parameter values within a specified parameter distribution in order to assess the distributions for response functions. Probability of event occurrence (e.g., failure) is then assessed by comparing the response results against response thresholds. DAKOTA currently implements Monte Carlo methods within the **NonDProbability** class.

The number of samples to be evaluated is selected with the `observations` integer specification. The `seed` integer specification specifies the seed for the random number generator which is used to make Monte Carlo studies repeatable. The parameter samples can be selected with pure Monte Carlo (by specifying `sample_type random`) or with latin hypercube Monte Carlo (by specifying `sample_type lhs`). Lastly, the `response_thresholds` specification supplies a list of `m` real values for comparison with the `m` response functions being computed. Table 50 provides the specification detail for the Monte Carlo probability method.

Table 50 Specification detail for the Monte Carlo probability method

Description	Specification	Sample	Status	Default
Monte Carlo probability	({nond_probability} ...)	nond_probability	Required group	N/A
observations	{observations = <INTEGER>}	observations = 100	Required	N/A
random seed	[seed = <INTEGER>]	seed = 1	Optional	1
sample type	{sample_type} {random} | {lhs}	sample_type, lhs	Required	N/A
response_thresholds	{response_thresholds = <LISTof> <REAL>}	response_thresholds = 1.0, 2.0	Required	N/A

Mean Value Method

The mean value method is selected using the `nond_mean_value` specification. This iterator computes approximate response function distribution statistics based on specified parameter distributions. The mean value method is a direct method and does not perform any random sampling.

The `response_filenames` specification supplies a list of file name strings for response data files which the mean value algorithm will process to determine the failure probability.

The specifics of this computation within the mean value implementation are currently application-dependent, but generalization is a pending development item. Table 51 provides the specification detail for the mean value method.

Table 51 Specification detail for the mean value method

Description	Specification	Sample	Status	Default
Mean value method	({nond_mean_value} ...)	nond_mean_value	Required group	N/A
response filenames	{response_filenames = <LISTof> <STRING>}	response_filenames = ‘r1.dat’, ‘r2.dat’	Required	N/A

Parameter Study Methods

DAKOTA’s parameter study methods compute response data sets at a selection of points in the parameter space. These points may be specified as a vector, a list, a set of centered vectors, or an n-dimensional hyper-surface. DAKOTA implements all of the parameter study methods within the **ParamStudy** class.

DAKOTA’s parameter study methods do not currently make use of the method independent controls for `max_iterations`, `max_function_evaluations`, `convergence_tolerance`, `speculative` gradients, `output` verbosity, or `linear_constraints`. Since each of the parameter study methods is consistent in this way, the parameter study documentation which follows is limited to the method dependent controls for the vector, list, centered, and multidimensional parameter study methods.

Capability overviews and examples of the different types of parameter studies are provided in Parameter Study Capabilities on page 62. The following discussions focus on the details of command specification.

Vector Parameter Study

DAKOTA’s vector parameter study computes response data sets at selected intervals along a vector in parameter space. It encompasses both single-coordinate parameter studies (to study the

effect of a single variable on a response set) and multiple coordinate vector studies (to investigate the response variations along some n-dimensional vector). This study is selected using the `vector_parameter_study` specification followed by either a `final_point` or a `step_vector` specification.

The vector for the study can be defined in several ways. First, a `final_point` specification, when combined with the Initial Values (see Initial Values on page 63), uniquely defines an n-dimensional vector's direction and magnitude through its start and end points. The intervals along this vector may either be specified with a `step_length` or a `num_steps` specification. In the former case, steps of equal length (Cartesian distance) are taken from the Initial Values up to (but not past) the `final_point`. The study will terminate at the last full step which does not go beyond the `final_point`. In the latter `num_steps` case, the distance between the Initial Values and the `final_point` is broken into `num_steps` intervals of equal length. This study starts at the Initial Values and ends at the `final_point`, making the total number of simulations equal to `num_steps+1`. The `final_point` specification detail is given in Table 52.

Table 52 final_point specification detail for the vector parameter study

Description	Specification	Sample	Status	Default
Vector parameter study	({vector_parameter_study} ...)	vector_parameter_ study	Required group	N/A
Final point with step length or number of steps	({final_point = <LISTof><REAL>} {step_length = <REAL>} | {num_steps = <INTEGER>})	final_point = 1.0,2.0 num_steps = 10	Required group	N/A

The other technique for defining a vector in the study is the `step_vector` specification. This parameter study starts at the Initial Values and adds the increments specified in `step_vector` to obtain new simulation points. This process is performed `num_steps` times, and since the Initial Values are included, the total number of simulations is again equal to `num_steps+1`. The `step_vector` specification detail is given in Table 53.

Table 53 step_vector specification detail for the vector parameter study

Description	Specification	Sample	Status	Default
Vector parameter study	({vector_parameter_study} ...)	vector_parameter_ study	Required group	N/A
Step vector and number of steps	({step_vector = <LISTof><REAL>} {num_steps = <INTEGER>})	step_vector = 1., 1., 1. num_steps = 10	Required group	N/A

Refer to Vector Parameter Study on page 63 for example specifications and the function evaluations that result.

List Parameter Study

DAKOTA's list parameter study allows for evaluations at user selected points of interest which need not be colinear or coplanar. This study is selected using the `list_parameter_study` method specification followed by a `list_of_points` specification.

The number of real values in the `list_of_points` specification must be a multiple of the total number of continuous variables specified in the variables section. This parameter study simply performs simulations for the first parameter set (the first `n` entries in the list), followed by the next parameter set (the next `n` entries), and so on, until the list of points has been exhausted. Since the Initial Values will not be used, they need not be specified. The list parameter study specification detail is given in Table 54.

Table 54 Specification detail for the list parameter study

Description	Specification	Sample	Status	Default
List parameter study	({list_parameter_study } ...)	list_parameter_study	Required group	N/A
List of points	{list_of_points = <LISTof> <REAL>}	list_of_points = 0.0, 0.0, 0.5, 0.0, 0.5, 0.5, 0.0, 0.5	Required	N/A

The sample `list_of_points` specification shown in Table 54 would perform simulations at the 4 corners of a square with edge length of 0.5 for a set of 2 variables.

Centered Parameter Study

DAKOTA's centered parameter study computes response data sets along multiple vectors, one per parameter, centered about the specified Initial Values. This is useful for investigation of function contours with respect to each parameter individually in the vicinity of a specific point (e.g., post-optimality analysis for verification of a minimum). It is selected using the `centered_parameter_study` method specification followed by `percent_delta` and `deltas_per_variable` specifications, where `percent_delta` specifies the size of the increments in percent and `deltas_per_variable` specifies the number of increments per variable in each of the plus and minus directions. The centered parameter study specification detail is given in Table 55.

Table 55 Specification detail for the centered parameter study

Description	Specification	Sample	Status	Default
Centered parameter study	({centered_parameter_study} ...)	centered_parameter_study	Required group	N/A
Interval size in percent	{percent_delta = <REAL>}	percent_delta = 1.0	Required	N/A
Number of +/- deltas per variable	{deltas_per_variable = <INTEGER>}	deltas_per_variable = 5	Required	N/A

Refer to Centered Parameter Study on page 66 for example specifications and the function evaluations that result.

Multidimensional Parameter Study

DAKOTA's multidimensional parameter study computes response data sets for an n-dimensional hypergrid of points. Each continuous variable is partitioned into equally spaced intervals between its upper and lower bounds, and each combination of the values defined by the boundaries of these partitions is evaluated. This study is selected using the `multidim_parameter_study` method specification followed by a `partitions` specification, where the partitions list specifies the number of partitions for each continuous variable. Therefore, the number of entries in the partitions list must be equal to the total number of continuous variables specified in the variables section. Since the Initial Values will not be used, they need not be specified. The multidimensional parameter study specification detail is given in Table 56.

Table 56 Specification detail for the multidimensional parameter study

Description	Specification	Sample	Status	Default
Multidimensional parameter study	({multidim_parameter_study} ...)	multidim_parameter_study	Required group	N/A
Partitions per variable	{partitions = <LISTof> <INTEGER>}	partitions = 4 2 4	Required	N/A

Refer to Multidimensional Parameter Study on page 67 for example specifications and the function evaluations that result.

Installation Guide

Distributions and Checkouts

Installation of DAKOTA can be done from a distribution file (tape, CD, secure Web site download, etc.) or a checkout from the Concurrent Version System (CVS) repository.

If you are extracting DAKOTA from a distribution file, first extract the distribution (`Dakota.tar.gz`) from the tape/CD/Web and move it to your installation directory. Then the following steps are performed:

```
gunzip Dakota.tar.gz
tar xvf Dakota.tar
```

If you are accessing current files from the CVS repository, you first need to have access to the CVS software on your workstation. You can get CVS via anonymous ftp from a number of sites, for instance, `prep.ai.mit.edu` in directory `pub/gnu`. Next, you need to be in the `dakota` developers' group and have your `$CVSROOT` environment variable set to the repository directory where DAKOTA resides (i.e., `/usr/local/eng_sci/CVS`). If, in addition, you are using the remote client-server capabilities of CVS, then the `$CVSROOT` variable needs a machine prefix (i.e., `sass2248:/usr/local/eng_sci/CVS`) and the `$CVS_RSH` environment variable must specify the remote shell program to use (e.g., `rsh`, `ssh`). The following steps can then be executed to check out the repository:

```
newgrp dakota
cd $HOME
cvs checkout Dakota
```

Basic Installation

Now that the DAKOTA files have been checked out or extracted, the next step is to configure and build the system using the following steps:

```
1) setenv DAKOTA $HOME/Dakota
2) cd $DAKOTA
3) ln -s <RogueWaveInstallationDir> rogue
4) ln -s <MPI_InstallationDir> mpi
5) configure <config_options>
6) make
```

Omission of step 1 is a common error; therefore it is wise to set this environment variable in your `.cshrc` file. Of course, `$DAKOTA` does not have to be set to `$HOME/Dakota`. If one wishes a different installation location or is maintaining multiple repositories or configurations of DAKOTA code, then `$DAKOTA` should be set and/or managed accordingly. This is in fact why the `$DAKOTA` variable exists.

The DAKOTA software relies upon the Rogue Wave Tools.h++ software, which is a C++ utility library for data management with vector classes, linked lists, hash tables, etc. If you are compiling on a Sun/Solaris host platform, this may be available as part of the C++ compiler distribution. If not, you will need to purchase a license for this product and install it on your workstation. Since there is no standard location for the Rogue Wave Tools.h++ software, the configure fragment files assume that the Rogue Wave software is installed in the directory `$DAKOTA/rogue`. Step 3 creates a symbolic link from this directory to the actual Rogue Wave installation directory.

To build DAKOTA with message-passing capability for parallel platforms, the MPI software must be installed on the target machine. There is no standard location for the MPI software (although `/usr/local/mpi` is common). Consequently, the configure fragment files assume that MPI is located in the directory `$DAKOTA/mpi`. Thus, step 4 creates a symbolic link from this directory to the actual MPI installation directory.

In both steps 3 and 4, the symbolic links must point to the directory level within the Rogue Wave and MPI distributions which contains the `bin`, `lib`, and `include` directories.

In step 5, the DAKOTA software is configured for building on specific hosts for specific target platforms. In the top-level directory defined by `$DAKOTA`, there exists a shell script called `configure` which is a program designed to automate much of the setup activity associated with building large suites of programs on various hardware platforms. Some of what `configure` does:

- makes symbolic links so that files used for configuration can be accessed from one location
- generates Makefiles so that objects, libraries, executables and other ‘targets’ can be created for specific and unique hardware platforms
- calls itself recursively so that sub-directories can also be configured

Refer to Configuration Details on page 181 and the Cygnus configure documentation (`$DAKOTA/docs/configure.ps`) for information on configure operations and options. Running configure without any options will result in inclusion of all vendor packages and exclusion of MPI.

In step 6, the Makefiles generated in the configure step are executed with the `make` command. Refer to Makefile Details on page 184 for additional information.

Configuration Details

The full parameter list for the `configure` script is below:

```
configure hosttype [--target=target] [--srcdir=dir] [--rm]
  [--site=site] [--prefix=dir] [--exec-prefix=dir]
  [--program-prefix=string] [--tmpdir=dir]
  [--with-package[=yes/no]] [--without-package]
  [--enable-feature[=yes/no]] [--disable-feature]
```

```
[--norecursion] [--nfp] [-s] [-v] [-V | - version]
[--help]
```

Makefiles are custom created from `Makefile.in` template files which outline the basic "targets" that can be built for each directory. Variables that are package, site or hardware dependent are stored in individual "fragment" files in the `$DAKOTA/config` directory. These fragment files are added to the custom Makefiles when users and code developers (recursively) configure this repository with specific host, target, package, and/or site parameters.

An example configuration command for a native build on a Sun/Solaris host using the SGOPT, DOT, NPSOL, and OPT++ vendor optimizer packages (see Configuring with specific vendor optimizers on page 183 for more info on packages) follows:

```
configure
```

NOTE: The `hosttype` and `--target` parameters are not necessary since available system information can be acquired from your local machine. If your Sun workstation is running Solaris 2.5.1, then the `config.guess` script will provide `configure` with the triplet '`sparc-sun-solaris2.5.1`'. If you wish to supply a `hosttype` parameter for a Sun/Solaris system, '`sun4sol2`' is preferred.

Running `configure` takes a while, be patient. Verbose output will always be displayed unless the user/developer wishes to silence it by specifying the parameter, `--silent`. If you wish to configure only one level/directory, please remember to use the option `--norecursion`. All generated `config.status` files include this parameter as a default for easy Makefile regeneration.

After your configure command is completed, three files will be generated in each configured directory (specified by the file `configure.in`).

1. `Makefile.${target_vendor}`

   The `${target_vendor}` suffix will depend on the target specified (i.e., "`sun`" for the command above). Native builds have identical host and target vendor values. If you specified a "`--target=tflop`" parameter, then `Makefile.intel` files would then be created for a cross-compilation build on the Solaris host for the Sandia Intel TFLOP (i.e., `janus`) target platform.

2. `Makefile`

   This will be a symbolic link to the file mentioned above. A user/developer will simply type "`make`" and the last generated `Makefile.${target_vendor}` will then be referenced.

3. `config.status`

   This is a "recording" of the configuration process (i.e., what commands were executed to generate the Makefile). It can be used by the custom Makefile to regenerate the configuration with the "`make Makefile`" command.

Fragment files exist so that `configure` can support multi-platform environments. DAKOTA can be configured for code development and execution on the following platforms :

```
SPARC-SUN-SOLARIS2.5.1or higher (i.e., Sun ULTRAsparc)
MIPS-SGI-IRIX6.5or higher (i.e., SGI Octane)
HPPA1.1-HP-HPUX9.05or higher (i.e., HP 9000/700 series)
PENTIUM-INTEL-COUGARor higher (i.e., Intel TFLOP
     supercomputer)
```

Below is a list of the fragment files used for configuring this software and examples of what dependent information they contain. They are listed in the order in which they will appear in the generated Makefiles. Inclusion of these fragment files is controlled by the `configure.in` file and any parameters you specify (i.e., `--with-<PACKAGE>` or `--target=<TGT_ALIAS>`) with the `configure` command.

- The following files contain package variables for location/definition of package source, include, library, defines, etc.

```
mp-opt++
mp-npsol
mp-dot-dp
mp-dot-sp
mp-sgopt
mp-stdlib
mp-mpi
mp-bayes
mp-cluster
mp-dakota
mp-idr
mp-twafer
```

- The following files contain target variables that help build Makefile targets (i.e., CC, CCC, AR, LEX, ARCH_DEFINES, ARCH_INCLUDES, ARCH_LIBS, etc.)

```
mt-solaris
mt-irix
mt-hpux
mt-cougar
```

- The following files contain host variables for administration/management of Makefile targets (i.e., AWK, CHMOD, RM, MKDIR, CD, etc.)

```
mh-solaris
mh-irix
mh-hpux
mh-cougar
```

- The following file contains site variables and macros for overriding implicit Makefile rules when building objects, archives, etc. It is always included by default in every generated Makefile unless overridden by a parameter (`--site=...`) to `configure`.

```
ms-dakota.std
```

Configuring with specific vendor optimizers

All of the available vendor optimizers (DOT, NPSOL, OPT++, and SGOPT) are configured for building by default. If the user/developer wishes to configure DAKOTA without any of the vendor optimizer packages, he/she must specify any combination of the following parameters: `--without-dot`, `--without-npsol`, `--without-optpp`, or `--without-sgopt`. Some examples follow:

- configure --without-npsol --without-sgopt

  Configure and generate Makefiles that construct an executable using libraries from the DOT and OPT++ optimizers **only**.
- configure --without-opt++

  Configure and generate Makefiles that construct an executable using libraries from the DOT, NPSOL, and SGOPT optimizers **only**.

Each of the configured vendor optimizer packages will contain their own individual ‘build’ directories. See Makefile Details on page 184 for more information concerning build directories and how they manage multi-platform binaries.

Configuring with the Message Passing Interface

The Message Passing Interface (MPI) package will not be configured into DAKOTA as a default unless the user configures for the Intel TFLOP target. If the user wishes to use this message-passing library on parallel platforms other than the Intel TFLOP distributed memory supercomputer, then `--with-mpi` must be specified. If the user configures for the Intel TFLOP target and does **not** wish to use MPI, then `--without-mpi` must be specified. Refer to Master-slave algorithm on page 103 for more information about the use of MPI within DAKOTA. Several examples follow:

- configure --target=tflop

  Configure and generate Makefiles that construct an executable for the Intel TFLOP platform using libraries from the DOT, NPSOL, SGOPT, and OPT++ optimizers **and** the MPI software package.
- configure --with-mpi

  Configure and generate Makefiles that construct an executable on your native platform (i.e., Solaris) using libraries from the DOT, NPSOL, SGOPT and OPT++ optimizers **and** the MPI software package.
- configure --target=tflop --without-mpi --without-sgopt

  Configure and generate Makefiles that construct an executable for the Intel TFLOP platform using libraries from the DOT, NPSOL and OPT++ optimizers **only**.

Makefile Details

Some versions of `make` fail to build the system properly. The `make` program in `/usr/ccs/bin` is preferred to `/usr/local/bin/make` on the Sun platform, and `gmake` is often preferred on other platforms. The version of `make` invoked by default can be queried by executing the command:

```
which make
```

If this is not the desired `make`, then the `$path` environment variable can be modified as in the following:

```
set path = (/usr/ccs/bin $path)
```

As with the `$DAKOTA` environment variable, it may be desirable to add this `$path` addition to the `.cshrc` file to render the change permanent.

As stated in Basic Installation on page 180, building/compiling the system after a successful configuration entails invoking the command "`make`" from the top-level `$DAKOTA` directory. The latest `Makefile.${target_vendor}` generated by `configure` will be referenced by this command (due to the `Makefile` symbolic link). Please note that build directories are generated to store object/library files and binaries for a particular target platform. If you configured DAKOTA for a native build on a Sun/Solaris host, your build directories will all be called `sparc-sun-solaris2.5.1`. If you configured DAKOTA for the Intel TFLOP platform, your build directories will all be called, `pentium-intel-cougar`.

During an initial `make` process, every makefile generates dependencies for the source files in the makefile's directory prior to actually compiling the object files and linking the libraries and/or executables. These dependencies are appended to the bottom of the makefiles and are used for managing which source files must be recompiled whenever header files are modified. This is needed because, while a standard makefile manages the dependency of targets on source files, it does not manage the dependence of source files on header files. If a developer changes the source file dependencies (e.g., by adding or removing `#include` directives), a "`make Makefile`" command can be used to create a fresh makefile and then a "`make`" command will create an updated dependency list, append the dependency list to the new makefile, and then recompile only the affected source modules.

You can remove object files, libraries, and executables from the build directories by typing "`make clean`". The `clean` target will also cause regeneration of dependencies. If you wish to reconfigure your DAKOTA source from scratch or regenerate all custom makefiles, "`make distclean`" can be used to remove all symbolic links, custom makefiles, and `config.status` files. Once in this state, the system must be reconfigured prior to building.

Each set of target "build" directories (and the object/library files and binaries they contain) is an independent entity. After configuring and building DAKOTA for a Sun/Solaris target, you can configure and build for, say, an Intel/TFLOP target, without destroying any previous Sun/Solaris files. Only the `Makefile` symbolic links are overwritten. Specific target binaries and object/library files get removed with a "`clean`" rule and specific target/build directories get removed with a "`distclean`" rule. Thus, a cleaning operation for one platform will not interfere with other platform files. However, multiple "`clean`" or "`distclean`" executions may be needed for each target platform in order to completely clean a distribution.

After a successful build, the actual "`dakota`" executable is located in the build directory within `$DAKOTA/src` (e.g., `$DAKOTA/src/sparc-sun-solaris2.5.1/dakota`). In addition, test simulator executables reside in the build directory within `$DAKOTA/test` (e.g.,

$DAKOTA/test/sparc-sun-solaris2.5.1/text_book). Symbolic links to these executables are provided in the $DAKOTA/test directory for testing convenience.

Caveats

Intel cross-compilation

The Intel iCC compilers provided by the Portland Group for the Cougar operating system require that the object and template instantiation files reside in the same directory as the source files for linking of the dakota executable. Therefore, the variable specifying the build directory in the dakota source (nominally $DAKOTA/src/pentium-intel-cougar) must be overridden in the source Makefile ($DAKOTA/src/Makefile.intel) to ensure that the objects are placed in the source directory, rather than a build subdirectory. To perform this override, two modifications must be made to Makefile.intel. Change the line:

```
DAKOTA_SRC_BUILD = $(DAKOTA_SRC)/$(target_canonical)
```

to

```
DAKOTA_SRC_BUILD = $(DAKOTA_SRC)
```

and then remove or comment out the following line from the distclean target in order to prevent removal of the source directory on a "make distclean":

```
$(RM) -r $(DAKOTA_SRC_BUILD)
```

System modifications

If you need to do unusual things to build this system, please determine if configure can be used to accomplish them. Notify us via e-mail by sending instructions to the address shown below so that a future release can incorporate your recommendations.

```
Michael S. Eldred, Sandia National Laboratories,
     mseldre@sandia.gov
```

Installation Examples

Sun Solaris platform

After checking out the repository or extracting the tape archive, a Dakota directory will be present which is ready for configuration and compilation. An example configuration performed on the Sandia JAL LAN is supplied in which the Dakota directory has been installed at the top level of a user directory.

First, one sets environment variables, changes directories to the correct directory for configuring and building, and installs soft links to the Rogue Wave Tools.h++ and MPI installation directories, e.g.:

```
setenv DAKOTA $HOME/Dakota
cd $DAKOTA
ln -s /usr/sharelan/dakota/rogue_wave/rogue rogue
ln -s /usr/local/mpi mpi
```

From this directory, executing the command

```
./configure --with-mpi
```

gives the following output with omissions as marked:

```
Configuring for a sparc-sun-solaris2.5.1 host.
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota using "config/mh-solaris" and
"config/mt-solaris" and "./config/ms-dakota.std"
Configuring idr...
Linked "config" to "./../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/idr using "config/mh-solaris" and
"config/mt-solaris" and "./config/ms-dakota.std"
Configuring VendorOptimizers...
Linked "config" to "./../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/VendorOptimizers using "config/
mh-solaris" and "config/mt-solaris" and "./config/ms-dakota.std"
Configuring sgopt...
sparc-sun-solaris2.5.1
Host/Target/Site Configuration:
  HOST          solaris
  TARGET        solaris
  SITE          dakota.std
  COMPILER
config/mp-solaris-dakota.std does not exist! Using a default configuration!
Package Configuration:
  MPI           no
  TCC           no
  GM            no
  COBYLA        no
  OPTIMIZATION  <default>
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/VendorOptimizers/sgopt using
"config/mf-solaris-solaris-dakota.std" and "./config/ms-dakota.std"
<<omission of SGOPT subdirectories>>
Configuring DOT...
Linked "config" to "./../../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/VendorOptimizers/DOT using
"config/mh-solaris" and "config/mt-solaris" and "./config/ms-dakota.std"
Configuring NPSOL...
```

```
Linked "config" to "./../../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/VendorOptimizers/NPSOL using
"config/mh-solaris" and "config/mt-solaris" and "./config/ms-dakota.std"
Configuring opt++...
Linked "config" to "./../../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/VendorOptimizers/opt++ using
"config/mh-solaris" and "config/mt-solaris" and "./config/ms-dakota.std"
<<omission of OPT++ subdirectories>>
Configuring src...
Linked "config" to "./../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/src using "config/mh-solaris" and
"config/mt-solaris" and "./config/ms-dakota.std"
Configuring test...
Linked "config" to "./../config".
Linked "Makefile" to "./Makefile.sun".
Created "Makefile.sun" in /home/mseldre/Dakota/test using "config/mh-solaris"
and "config/mt-solaris" and "./config/ms-dakota.std"
```

as it generates Makefiles in the DAKOTA subdirectories.

Now that Makefiles have been created, executing the command

```
make
```

from the same `$DAKOTA` directory will build the system. While this output is too lengthy to fully replicate here, some excerpts are provided below with omissions as marked:

```
============================================================
= Building Input Deck Reader executable: 'idrtest' - BEGIN =
============================================================
if [ ! -d $DAKOTA/idr/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/idr/sparc-sun-solaris2.5.1; \
fi
/usr/ccs/bin/make -f Makefile.sun $DAKOTA/idr/sparc-sun-solaris2.5.1/idrtest

<<omission>>

============================================================
= Building Input Deck Reader executable: 'idrtest' - END =
============================================================


================================================
= Install DAKOTA software - BEGIN  =
================================================
============================================================
= Install Input Deck Reader library - BEGIN =
============================================================
if [ ! -d $DAKOTA/idr/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/idr/sparc-sun-solaris2.5.1; \
fi
/usr/ccs/bin/make -f Makefile.sun library

Archiving Object File(s) --  idr.o idr-parser.o

ar ru $DAKOTA/idr/sparc-sun-solaris2.5.1/libidr.a $DAKOTA/idr/sparc-sun-
solaris2.5.1/idr.o $DAKOTA/idr/sparc-sun-solaris2.5.1/idr-parser.o
ar: creating $DAKOTA/idr/sparc-sun-solaris2.5.1/libidr.a
ls -lF $DAKOTA/idr/sparc-sun-solaris2.5.1/libidr.a
-rw-rw-r--   1 <user>   <user>     46684 Jan  8 09:52 $DAKOTA/idr/sparc-sun-
solaris2.5.1/libidr.a

============================================================
= Install Input Deck Reader library - END =
============================================================
============================================================
= Install DAKOTA VendorOptimizers - BEGIN  =
============================================================
```

```
(for DIRS in sgopt DOT NPSOL opt++; do  \
  cd ${DIRS}; /usr/ccs/bin/make -f Makefile.sun install; cd ..; \
done)
==========================================
= Install SGOPT Software - BEGIN =
==========================================
if [ ! -d $DAKOTA/VendorOptimizers/sgopt/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/VendorOptimizers/sgopt/sparc-sun-solaris2.5.1; \
fi
(for DIRS in packages src examples; do  \
  cd ${DIRS}; /usr/ccs/bin/make -f Makefile.sun install; cd ..; \
done)
```

<<omission>>

```
==========================================
= Install SGOPT Software - END =
==========================================
====================================
= Install DOT Package - BEGIN =
====================================
if [ ! -d $DAKOTA/VendorOptimizers/DOT/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/VendorOptimizers/DOT/sparc-sun-solaris2.5.1; \
fi
(/usr/ccs/bin/make -f Makefile.sun library);
```

<<omission>>

```
====================================
= Install DOT Package - END   =
====================================
====================================
= Install NPSOL Package - BEGIN =
====================================
if [ ! -d $DAKOTA/VendorOptimizers/NPSOL/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/VendorOptimizers/NPSOL/sparc-sun-solaris2.5.1; \
fi
(/usr/ccs/bin/make -f Makefile.sun library);
```

<<omission>>

```
====================================
= Install NPSOL Package - END   =
====================================
========================================
= Install OPT++ Package - BEGIN =
========================================
if [ ! -d $DAKOTA/VendorOptimizers/opt++/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/VendorOptimizers/opt++/sparc-sun-solaris2.5.1; \
fi
```

<<omission>>

```
========================================
= Install OPT++ Package - END  =
========================================
=========================================================
= Install DAKOTA VendorOptimizers - END  =
=========================================================
==========================================
= Install DAKOTA Source - BEGIN =
==========================================
if [ ! -d $DAKOTA/src/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/src/sparc-sun-solaris2.5.1; \
fi
(/usr/ccs/bin/make -f Makefile.sun $DAKOTA/src/sparc-sun-solaris2.5.1/
libdakota.a);
```

<<omission>>

```
(/usr/ccs/bin/make -f Makefile.sun $DAKOTA/src/sparc-sun-solaris2.5.1/dakota);
```

```
Linking Object File(s) -- Creating DAKOTA executable: dakota

CC -fast   -D__EXTERN_C__ -DDAKOTA_SGOPT -DDAKOTA_DOT -DDAKOTA_NPSOL -
DDAKOTA_OPTPP -DNEWMAT -DSERIAL -DUNIX -DSOLARIS -DMULTITASK -I$DAKOTA/src/. -
I$DAKOTA/idr/. -I$DAKOTA/VendorOptimizers/sgopt/include/.  -I$DAKOTA/
VendorOptimizers/sgopt/packages/stdlib/include/. -I$DAKOTA/VendorOptimizers/DOT/
include/. -I$DAKOTA/VendorOptimizers/NPSOL/include/. -I$DAKOTA/VendorOptimizers/
opt++/include/.   -L/opt/SUNWspro/SC4.2/lib -o $DAKOTA/src/sparc-sun-
solaris2.5.1/dakota $DAKOTA/src/sparc-sun-solaris2.5.1/main.o
$DAKOTA/src/sparc-sun-solaris2.5.1/decomp.o                    $DAKOTA/src/
sparc-sun-solaris2.5.1/init_parallel_lib.o         $DAKOTA/src/sparc-sun-
solaris2.5.1/keywordtable.o             $DAKOTA/src/sparc-sun-solaris2.5.1/
CommandLineHandler.o    $DAKOTA/src/sparc-sun-solaris2.5.1/DakotaModel.o
$DAKOTA/src/sparc-sun-solaris2.5.1/DakotaVariables.o           $DAKOTA/src/
sparc-sun-solaris2.5.1/DakotaResponse.o            $DAKOTA/src/sparc-sun-
solaris2.5.1/DakotaInterface.o          $DAKOTA/src/sparc-sun-solaris2.5.1/
ApplicationInterface.o      $DAKOTA/src/sparc-sun-solaris2.5.1/
SysCallApplicInterface.o    $DAKOTA/src/sparc-sun-solaris2.5.1/
DirectFnApplicInterface.o   $DAKOTA/src/sparc-sun-solaris2.5.1/
DirectFnTextBook.o          $DAKOTA/src/sparc-sun-solaris2.5.1/
ExecutableProgram.o     $DAKOTA/src/sparc-sun-solaris2.5.1/AnalysisCode.o
$DAKOTA/src/sparc-sun-solaris2.5.1/CommandShell.o              $DAKOTA/src/
sparc-sun-solaris2.5.1/ParamResponsePair.o         $DAKOTA/src/sparc-sun-
solaris2.5.1/ProblemDescDB.o            $DAKOTA/src/sparc-sun-solaris2.5.1/
DataMethod.o          $DAKOTA/src/sparc-sun-solaris2.5.1/DataVariables.o
$DAKOTA/src/sparc-sun-solaris2.5.1/DataResponses.o             $DAKOTA/src/
sparc-sun-solaris2.5.1/DataInterface.o             $DAKOTA/src/sparc-sun-
solaris2.5.1/DakotaStrategy.o           $DAKOTA/src/sparc-sun-solaris2.5.1/
SingleMethodStrategy.o      $DAKOTA/src/sparc-sun-solaris2.5.1/
MultilevelOptStrategy.o     $DAKOTA/src/sparc-sun-solaris2.5.1/
SeqApproxOptStrategy.o  $DAKOTA/src/sparc-sun-solaris2.5.1/NonDOptStrategy.o
$DAKOTA/src/sparc-sun-solaris2.5.1/DakotaIterator.o            $DAKOTA/src/
sparc-sun-solaris2.5.1/ParamStudy.o                $DAKOTA/src/sparc-sun-
solaris2.5.1/DakotaNonD.o               $DAKOTA/src/sparc-sun-solaris2.5.1/
NonDProbability.o      $DAKOTA/src/sparc-sun-solaris2.5.1/NonDMeanValue.o
$DAKOTA/src/sparc-sun-solaris2.5.1/Lhs.o                       $DAKOTA/src/
sparc-sun-solaris2.5.1/LhsInput.o                  $DAKOTA/src/sparc-sun-
solaris2.5.1/Vm_util.o                  $DAKOTA/src/sparc-sun-solaris2.5.1/
DakotaOptimizer.o    $DAKOTA/src/sparc-sun-solaris2.5.1/DOTOptimizer.o
$DAKOTA/src/sparc-sun-solaris2.5.1/SNLLOptimizer.o
$DAKOTA/src/sparc-sun-solaris2.5.1/SGOPTOptimizer.o  $DAKOTA/src/sparc-sun-
solaris2.5.1/SGOPTRealApplication.o                              $DAKOTA/src/
sparc-sun-solaris2.5.1/NPSOLOptimizer.o  $DAKOTA/src/sparc-sun-solaris2.5.1/
npoptn_wrapper.o $DAKOTA/idr/sparc-sun-solaris2.5.1/libidr.a $DAKOTA/
VendorOptimizers/sgopt/sparc-sun-solaris2.5.1/libsgopt.a $DAKOTA/
VendorOptimizers/sgopt/packages/stdlib/sparc-sun-solaris2.5.1/libstdlib.a
$DAKOTA/VendorOptimizers/DOT/sparc-sun-solaris2.5.1/libdot.a $DAKOTA/
VendorOptimizers/NPSOL/sparc-sun-solaris2.5.1/libnpsol.a $DAKOTA/
VendorOptimizers/opt++/sparc-sun-solaris2.5.1/liboptpp.a -lrwtool -lM77 -lF77 -
lsunmath -ll -ly -lm

===========================================
= Install DAKOTA Source - END   =
===========================================
=================================================
= Install DAKOTA Test code - BEGIN =
=================================================
if [ ! -d $DAKOTA/test/sparc-sun-solaris2.5.1 ]; then \
   mkdir -m 775 $DAKOTA/test/sparc-sun-solaris2.5.1; \
fi

<<omission>>

=================================================
= Install DAKOTA Test code - END =
=================================================
===========================================
= Install DAKOTA software - END =
===========================================
```

You can now change directories to the test area

```
cd test
```

and execute dakota on the test files therein, e.g.:

```
dakota -i dakota_textbook.in
```

Textbook Example

Textbook Problem Formulation

The optimization problem formulation is stated as

`minimize`

$$f = (x_1 - 1)^4 + (x_2 - 1)^4 + \ldots + (x_n - 1)^4 \tag{8}$$

`subject to`

$$g_1 = x_1^2 - \frac{x_2}{2} \leq 0 \tag{9}$$

$$g_2 = x_2^2 - 0.5 \leq 0 \tag{10}$$

$$0.5 \leq x_1 \leq 5.8 \tag{11}$$

$$-2.9 \leq x_2 \leq 2.9 \tag{12}$$

This example problem may also be used to exercise least squares solution methods by modifying the problem formulation to:

`minimize`

$$(f)^2 + (g_1)^2 + (g_2)^2 \tag{13}$$

This modification is performed by simply changing the responses specification for the three functions from `num_objective_functions = 1` and `num_nonlinear_constraints = 2` to `num_least_squares_terms = 3`. Note that the 2 problem formulations are not equivalent and will have different solutions. More specifically, the optimization solution seeks to find the minimum objective function which satisfies the constraint inequalities, whereas the least squares formulation seeks to minimize the sum of the squares of the three residual functions.

Another way to exercise the least squares methods which would be equivalent to the optimization formulation would be to select the residual functions to be $(x_i\text{-}1)^2$. However, this formulation requires significant modification to `text_book.C` and will not be presented here. Equation (13), on the other hand, does not require any modification to `text_book.C`. Refer to Rosenbrock Example on page 204 for an example of minimizing the same objective function using both optimization and least squares approaches.

Methods

DOT and NPSOL methods may be used to solve this optimization problem with or without the constraints. OPT++ methods may be used to solve the unconstrained optimization problem or the least squares problem.

The `dakota_textbook.in` file provided in the `Dakota/test` directory selects a `dot_mmfd` optimizer to perform constrained minimization using the `text_book` simulator. This simulator returns analytic gradients as requested by the optimizer.

A multilevel hybrid can also be demonstrated on the `text_book` problem. The `dakota_multilevel.in` file provided in the `Dakota/test` directory starts with a `sgopt_pga_real` solution which feeds its best point into a `sgopt_coord_sps` optimization which feeds its best point into `optpp_newton`. While this approach is overkill for such a simple problem, it is useful for demonstrating the coordination between multiple methods in the multilevel strategy.

In addition, `dakota_textbook_3pc.in` demonstrates the use of a 3-piece interface to perform the parameter to response mapping and `dakota_textbook_lhs.in` demonstrates the use of latin hypercube Monte Carlo sampling for assessing probability of failure as measured by specified response thresholds.

Results

Optimization

The solution for the unconstrained optimization problem for 2 design variables is:

$x_1 = 1.0$
$x_2 = 1.0$

with

$f^* = 0.0$

The solution for the optimization problem constrained by g_1 is:

$x_1 = 0.763$
$x_2 = 1.16$

with

$f^* = 0.00388$
$g_1^* = 0.0$ (active)

The solution for the optimization problem constrained by g_1 and g_2 is:

$x_1 = 0.594$
$x_2 = 0.707$

with

$f^* = 0.0345$
$g_1^* = 0.0$ (active)
$g_2^* = 0.0$ (active)

Note that as constraints are added, the design freedom is restricted and a penalty in the objective function is observed. Of course, no penalty would be observed if the additional constraints were not active at the solution.

The `dot_sqp` optimizer navigates to the constrained optimum in 12 function calls and 5 gradient calls (17 evaluations total). The output from this minimization is shown below:

```
MPI initialized with 1 processors.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
no method_pointer: last specifications parsed will be used
methodName = dot_sqp
gradientType = analytic
hessianType = none
Running MPI executable in serial mode.
Running Single Method Strategy...
```

```
DDDDD        OOOOO       TTTTTTT
D    D      O     O         T
D    D  ==  O  *  O  ==     T
D    D      O     O         T
DDDDD        OOOOO          T

   DESIGN OPTIMIZATION TOOLS


     (C) COPYRIGHT, 1995

             VR&D

ALL RIGHTS RESERVED, WORLDWIDE

          VERSION 4.20


- YOUR INTEGRITY IS OUR COPY PROTECTION -
```

```
  CONTROL PARAMETERS

  OPTIMIZATION METHOD,               METHOD =       3
  NUMBER OF DECISION VARIABLES,         NDV =       2
  NUMBER OF CONSTRAINTS,               NCON =       2
  PRINT CONTROL PARAMETER,           IPRINT =       3
  GRADIENT PARAMETER,                 IGRAD =       1
    GRADIENTS ARE SUPPLIED BY THE USER
  THE OBJECTIVE FUNCTION WILL BE MINIMIZED

------------------------------
Begin Function Evaluation    1
------------------------------
```

```
Parameters for function evaluation 1:
                     9.0000000000e-01 x1
                     1.1000000000e+00 x2

(text_book text_book.in.1 text_book.out.1)

Active response data for function evaluation 1:
Active set vector = { 1 1 1 }
                     2.0000000000e-04 obj_fn
                     2.6000000000e-01 nln_con1
                     7.1000000000e-01 nln_con2



  -- SCALAR PROGRAM PARAMETERS

  REAL PARAMETERS
    1) CT     = -3.00000E-02            8) DX2    =  2.20000E-01
    2) CTMIN  =  3.00000E-03            9) FDCH   =  1.00000E-03
    3) DABOBJ =  2.00000E-08           10) FDCHM  =  1.00000E-04
    4) DELOBJ =  1.00000E-04           11) RMVLMZ =  4.00000E-01
    5) DOBJ1  =  1.00000E-01           12) DABSTR =  2.00000E-08
    6) DOBJ2  =  2.00000E-01           13) DELSTR =  1.00000E-03
    7) DX1    =  1.00000E-02

  INTEGER PARAMETERS
    1) IGRAD  =      1    6) NCOLA  =      2   11) IPRNT1 =      0
    2) ISCAL  =   1000    7) IGMAX  =      0   12) IPRNT2 =      0
    3) ITMAX  =     50    8) JTMAX  =     50   13) JWRITE =      0
    4) ITRMOP =      2    9) ITRMST =      2
    5) IWRITE =      6   10) JPRINT =      0


     STORAGE REQUIREMENTS
  ARRAY  DIMENSION  REQUIRED
    WK       136       136
   IWK        81        81


  -- INITIAL VARIABLES AND BOUNDS

  LOWER BOUNDS ON THE DECISION VARIABLES (XL-VECTOR)
       1)   5.00000E-01  -2.90000E+00

  DECISION VARIABLES (X-VECTOR)
       1)   9.00000E-01   1.10000E+00

  UPPER BOUNDS ON THE DECISION VARIABLES (XU-VECTOR)
       1)   5.80000E+00   2.90000E+00


  -- INITIAL FUNCTION VALUES

  OBJ =  2.00000E-04

  CONSTRAINT VALUES (G-VECTOR)
       1)   2.60000E-01   7.10000E-01


  -- BEGIN CONSTRAINED OPTIMIZATION: SQP METHOD


  -- BEGIN SQP ITERATION    1

------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                     9.0000000000e-01 x1
                     1.1000000000e+00 x2
```

```
(text_book text_book.in.2 text_book.out.2)

Active response data for function evaluation 2:
Active set vector = { 2 2 2 }
 [ -4.0000000000e-03  4.0000000000e-03  ] obj_fn gradient
 [  1.8000000000e+00 -5.0000000000e-01  ] nln_con1 gradient
 [  0.0000000000e+00  2.2000000000e+00  ] nln_con2 gradient



------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                      6.6386363636e-01 x1
                      7.7590909091e-01 x2

(text_book text_book.in.3 text_book.out.3)

Active response data for function evaluation 3:
Active set vector = { 1 1 1 }
                      1.5287930702e-02 obj_fn
                      5.2760382226e-02 nln_con1
                      1.0203491736e-01 nln_con2



------------------------------
Begin Function Evaluation    4
------------------------------
Duplication detected in response requests for this parameter set:
                      6.6386363636e-01 x1
                      7.7590909091e-01 x2

Active response data retrieved from database:
Active set vector = { 1 1 1 }
                      1.5287930702e-02 obj_fn
                      5.2760382226e-02 nln_con1
                      1.0203491736e-01 nln_con2



  OBJ = 1.52879E-02

  DECISION VARIABLES (X-VECTOR)
       1)   6.63864E-01   7.75909E-01

  CONSTRAINT VALUES (G-VECTOR)
       1)   5.27604E-02   1.02035E-01

  GMAX =  1.0203E-01


  -- BEGIN SQP ITERATION    2

------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                      6.6386363636e-01 x1
                      7.7590909091e-01 x2

(text_book text_book.in.5 text_book.out.5)

Active response data for function evaluation 5:
Active set vector = { 2 2 2 }
 [ -1.5191703790e-01 -4.5012455672e-02  ] obj_fn gradient
 [  1.3277272727e+00 -5.0000000000e-01  ] nln_con1 gradient
 [  0.0000000000e+00  1.5518181818e+00  ] nln_con2 gradient
```

```
------------------------------
Begin Function Evaluation    6
------------------------------
Parameters for function evaluation 6:
                      5.9637770195e-01 x1
                      7.0822402407e-01 x2

(text_book text_book.in.6 text_book.out.6)

Active response data for function evaluation 6:
Active set vector = { 1 1 1 }
                      3.3787645889e-02 obj_fn
                      1.5543513482e-03 nln_con1
                      1.5812682699e-03 nln_con2



------------------------------
Begin Function Evaluation    7
------------------------------
Parameters for function evaluation 7:
                      6.0987488883e-01 x1
                      7.2176103744e-01 x2

(text_book text_book.in.7 text_book.out.7)

Active response data for function evaluation 7:
Active set vector = { 1 1 1 }
                      2.9157489712e-02 obj_fn
                      1.1066861305e-02 nln_con1
                      2.0938995166e-02 nln_con2



------------------------------
Begin Function Evaluation    8
------------------------------
Parameters for function evaluation 8:
                      6.1399879055e-01 x1
                      7.2589710766e-01 x2

(text_book text_book.in.8 text_book.out.8)

Active response data for function evaluation 8:
Active set vector = { 1 1 1 }
                      2.7844963118e-02 obj_fn
                      1.4045960967e-02 nln_con1
                      2.6926610909e-02 nln_con2



  OBJ = 2.78450E-02

  DECISION VARIABLES (X-VECTOR)
       1)   6.13999E-01   7.25897E-01

  CONSTRAINT VALUES (G-VECTOR)
       1)   1.40460E-02   2.69266E-02

  GMAX =  2.6927E-02


  -- BEGIN SQP ITERATION    3

------------------------------
Begin Function Evaluation    9
------------------------------
Parameters for function evaluation 9:
                      6.1399879055e-01 x1
                      7.2589710766e-01 x2

(text_book text_book.in.9 text_book.out.9)
```

```
Active response data for function evaluation 9:
Active set vector = { 2 2 2 }
 [ -2.3005198645e-01 -8.2376027758e-02  ] obj_fn gradient
 [  1.2279975811e+00 -5.0000000000e-01  ] nln_con1 gradient
 [  0.0000000000e+00  1.4517942153e+00  ] nln_con2 gradient



------------------------------
Begin Function Evaluation   10
------------------------------
Parameters for function evaluation 10:
                     5.9089395588e-01 x1
                     7.0528357263e-01 x2

(text_book text_book.in.10 text_book.out.10)

Active response data for function evaluation 10:
Active set vector = { 1 1 1 }
                     3.5556238180e-02 obj_fn
                    -3.4861192195e-03 nln_con1
                    -2.5750821783e-03 nln_con2



------------------------------
Begin Function Evaluation   11
------------------------------
Parameters for function evaluation 11:
                     5.9551492281e-01 x1
                     7.0940627964e-01 x2

(text_book text_book.in.11 text_book.out.11)

Active response data for function evaluation 11:
Active set vector = { 1 1 1 }
                     3.3898544900e-02 obj_fn
                    -6.5116530600e-05 nln_con1
                     3.2572695927e-03 nln_con2



------------------------------
Begin Function Evaluation   12
------------------------------
Parameters for function evaluation 12:
                     5.9559270455e-01 x1
                     7.0947567449e-01 x2

(text_book text_book.in.12 text_book.out.12)

Active response data for function evaluation 12:
Active set vector = { 1 1 1 }
                     3.3871152259e-02 obj_fn
                    -7.1675318164e-06 nln_con1
                     3.3557326930e-03 nln_con2



  OBJ = 3.38712E-02

  DECISION VARIABLES (X-VECTOR)
       1)   5.95593E-01   7.09476E-01

  CONSTRAINT VALUES (G-VECTOR)
       1)  -7.16753E-06   3.35573E-03

  GMAX =  3.3557E-03


  -- BEGIN SQP ITERATION    4
```

```
------------------------------
Begin Function Evaluation   13
------------------------------
Parameters for function evaluation 13:
                     5.9559270455e-01 x1
                     7.0947567449e-01 x2

(text_book text_book.in.13 text_book.out.13)

Active response data for function evaluation 13:
Active set vector = { 2 2 2 }
 [ -2.6455558611e-01 -9.8086106593e-02  ] obj_fn gradient
 [  1.1911854091e+00 -5.0000000000e-01  ] nln_con1 gradient
 [  0.0000000000e+00  1.4189513490e+00  ] nln_con2 gradient



------------------------------
Begin Function Evaluation   14
------------------------------
Parameters for function evaluation 14:
                     5.9371257075e-01 x1
                     7.0499649858e-01 x2

(text_book text_book.in.14 text_book.out.14)

Active response data for function evaluation 14:
Active set vector = { 1 1 1 }
                      3.4821641822e-02 obj_fn
                     -3.6326234263e-06 nln_con1
                     -2.9799369899e-03 nln_con2



------------------------------
Begin Function Evaluation   15
------------------------------
Parameters for function evaluation 15:
                     5.9408859751e-01 x1
                     7.0589233377e-01 x2

(text_book text_book.in.15 text_book.out.15)

Active response data for function evaluation 15:
Active set vector = { 1 1 1 }
                      3.4629329912e-02 obj_fn
                     -4.9051936012e-06 nln_con1
                     -1.7160131247e-03 nln_con2



------------------------------
Begin Function Evaluation   16
------------------------------
Parameters for function evaluation 16:
                     5.9442052455e-01 x1
                     7.0668310706e-01 x2

(text_book text_book.in.16 text_book.out.16)

Active response data for function evaluation 16:
Active set vector = { 1 1 1 }
                      3.4460496673e-02 obj_fn
                     -5.7935237028e-06 nln_con1
                     -5.9898619602e-04 nln_con2



  OBJ = 3.44605E-02

  DECISION VARIABLES (X-VECTOR)
```

```
       1)   5.94421E-01   7.06683E-01

  CONSTRAINT VALUES (G-VECTOR)
       1)  -5.79352E-06  -5.98986E-04

  GMAX = -5.7935E-06


  -- BEGIN SQP ITERATION    5

------------------------------
Begin Function Evaluation   17
------------------------------
Parameters for function evaluation 17:
                     5.9442052455e-01 x1
                     7.0668310706e-01 x2

(text_book text_book.in.17 text_book.out.17)

Active response data for function evaluation 17:
Active set vector = { 2 2 2 }
 [ -2.6686271425e-01 -1.0094184051e-01  ] obj_fn gradient
 [  1.1888410491e+00 -5.0000000000e-01  ] nln_con1 gradient
 [  0.0000000000e+00  1.4133662141e+00  ] nln_con2 gradient



  Q.P. SUB-PROBLEM GAVE NULL SEARCH DIRECTION.  CONVERGENCE ASSUMED.



  -- OPTIMIZATION IS COMPLETE

  NUMBER OF CONSTRAINED MINIMIZATIONS =    5

  CONSTRAINT TOLERANCE, CT =-3.00000E-02

  THERE ARE       2 ACTIVE CONSTRAINTS AND       0 VIOLATED CONSTRAINTS
  CONSTRAINT NUMBERS
           1         2

  THERE ARE       0 ACTIVE SIDE CONSTRAINTS

  TERMINATION CRITERIA

  MAXIMUM S-VECTOR COMPONENT = 0.00000E+00 IS LESS THAN 1.00000E-04


  -- OPTIMIZATION RESULTS


  OBJECTIVE, F(X) =    3.44605E-02


  DECISION VARIABLES, X

     ID         XL             X            XU
      1     5.00000E-01   5.94421E-01   5.80000E+00
      2    -2.90000E+00   7.06683E-01   2.90000E+00


  CONSTRAINTS, G(X)

       1)  -5.79352E-06  -5.98986E-04

  FUNCTION CALLS =      12

  GRADIENT CALLS =       5

<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 17 total (16 new, 1 duplicate)
```

```
<<<<< Best design parameters  =
                      5.9442052455e-01 x1
                      7.0668310706e-01 x2
<<<<< Best objective function =
                      3.4460496673e-02
<<<<< Best constraint values  =
                     -5.7935237028e-06
                     -5.9898619602e-04
Run time from MPI_Init to MPI_Finalize is 2.3499540000e+00 seconds
```

Least Squares

The solution for the least squares problem is:

$x_1 = 0.602$
$x_2 = 0.710$

with the residual functions equal to

$f^* = 0.0322$
$g_1^* = 0.00673$
$g_2^* = 0.00455$

and a minimal sum of the squares of 0.00111.

This study requires selection of `num_least_squares_terms = 3` in the responses specification and selection of either `optpp_g_newton` or `optpp_bcg_newton` in the method specification. The `optpp_bcg_newton` method navigates to the least squares solution in 5 function and gradient calls. This output is shown below:

```
MPI initialized with 1 processors.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
no method_pointer: last specifications parsed will be used
methodName = optpp_bcg_newton
gradientType = analytic
hessianType = none
Running MPI executable in serial mode.
Running Single Method Strategy...

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      9.0000000000e-01 x1
                      1.1000000000e+00 x2

(text_book text_book.in.1 text_book.out.1)

Active response data for function evaluation 1:
Active set vector = { 3 3 3 }
                      2.0000000000e-04 least_sq_term1
                      2.6000000000e-01 least_sq_term2
                      7.1000000000e-01 least_sq_term3
 [ -4.0000000000e-03  4.0000000000e-03  ] least_sq_term1 gradient
 [  1.8000000000e+00 -5.0000000000e-01  ] least_sq_term2 gradient
 [  0.0000000000e+00  2.2000000000e+00  ] least_sq_term3 gradient


   nlf2_evaluator_gn results: objective fn. =
   5.7170004000e-01
   nlf2_evaluator_gn results: objective fn. gradient =
 [  9.3599840000e-01  2.8640016000e+00 ]
   nlf2_evaluator_gn results: objective fn. Hessian =
[[  6.4800320000e+00 -1.8000320000e+00
```

```
   -1.8000320000e+00  1.0180032000e+01 ]]


------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                     6.6590894007e-01 x1
                     7.7727283167e-01 x2

(text_book text_book.in.2 text_book.out.2)

Active response data for function evaluation 2:
Active set vector = { 3 3 3 }
                     1.4919211444e-02 least_sq_term1
                     5.4798300630e-02 least_sq_term2
                     1.0415305485e-01 least_sq_term3
 [ -1.4916074862e-01 -4.4195655359e-02  ] least_sq_term1 gradient
 [  1.3318178801e+00 -5.0000000000e-01  ] least_sq_term2 gradient
 [  0.0000000000e+00  1.5545456633e+00  ] least_sq_term3 gradient


    nlf2_evaluator_gn results: objective fn. =
    1.4073295457e-02
    nlf2_evaluator_gn results: objective fn. gradient =
 [  1.4151199166e-01  2.6770433019e-01 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  3.5919755894e+00 -1.3186333660e+00
   -1.3186333660e+00  5.3371309505e+00 ]]


------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                     6.0233226286e-01 x1
                     7.1140623577e-01 x2

(text_book text_book.in.3 text_book.out.3)

Active response data for function evaluation 3:
Active set vector = { 3 3 3 }
                     3.1944760199e-02 least_sq_term1
                     7.1010369970e-03 least_sq_term2
                     6.0988322924e-03 least_sq_term3
 [ -2.5154811392e-01 -9.6143697434e-02  ] least_sq_term1 gradient
 [  1.2046645257e+00 -5.0000000000e-01  ] least_sq_term2 gradient
 [  0.0000000000e+00  1.4228124715e+00  ] least_sq_term3 gradient


    nlf2_evaluator_gn results: objective fn. =
    1.1080881859e-03
    nlf2_evaluator_gn results: objective fn. gradient =
 [  1.0374463766e-03  4.1113775791e-03 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  3.0289861462e+00 -1.1562949942e+00
   -1.1562949942e+00  4.5672778792e+00 ]]


------------------------------
Begin Function Evaluation    4
------------------------------
Parameters for function evaluation 4:
                     6.0157271127e-01 x1
                     7.1031375941e-01 x2

(text_book text_book.in.4 text_book.out.4)

Active response data for function evaluation 4:
Active set vector = { 3 3 3 }
                     3.2242004707e-02 least_sq_term1
                     6.7328472397e-03 least_sq_term2
```

```
                     4.5456368072e-03 least_sq_term3
 [ -2.5299225122e-01 -9.7239696468e-02  ] least_sq_term1 gradient
 [  1.2031454225e+00 -5.0000000000e-01  ] least_sq_term2 gradient
 [  0.0000000000e+00  1.4206275188e+00  ] least_sq_term3 gradient


    nlf2_evaluator_gn results: objective fn. =
    1.1055409135e-03
    nlf2_evaluator_gn results: objective fn. gradient =
 [ -1.1276603567e-04 -8.7939264600e-05 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  3.0231279737e+00 -1.1539436431e+00
   -1.1539436431e+00  4.5552762115e+00 ]]


------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                     6.0162216282e-01 x1
                     7.1034559141e-01 x2

(text_book text_book.in.5 text_book.out.5)

Active response data for function evaluation 5:
Active set vector = { 3 3 3 }
                     3.2226401354e-02 least_sq_term1
                     6.7764310912e-03 least_sq_term2
                     4.5908592356e-03 least_sq_term3
 [ -2.5289806109e-01 -9.7207644612e-02  ] least_sq_term1 gradient
 [  1.2032443256e+00 -5.0000000000e-01  ] least_sq_term2 gradient
 [  0.0000000000e+00  1.4206911828e+00  ] least_sq_term3 gradient


    nlf2_evaluator_gn results: objective fn. =
    1.1055369511e-03
    nlf2_evaluator_gn results: objective fn. gradient =
 [  7.4156799421e-06  2.6502438991e-06 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  3.0235086728e+00 -1.1540770759e+00
   -1.1540770759e+00  4.5556255261e+00 ]]


<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 5 total (5 new, 0 duplicate)
<<<<< Best design parameters  =
                     6.0162216282e-01 x1
                     7.1034559141e-01 x2
<<<<< Best objective function =
                     1.1055369511e-03
Run time from MPI_Init to MPI_Finalize is 9.5173000000e-01 seconds
```

Rosenbrock Example

Rosenbrock Problem Formulation on page 204

Rosenbrock Problem Formulation

The Rosenbrock function (see [Gill, P.E., Murray, W., and Wright, M.H., 1981]) is a well known benchmark problem for optimization algorithms. Its formulation can be stated as

`minimize`

$$f = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{14}$$

This example problem may also be used to exercise least squares solution methods by recasting the problem formulation into:

`minimize`

$$f = (f_1)^2 + (f_2)^2 \tag{15}$$

where

$$f_1 = 10(x_2 - x_1^2) \tag{16}$$

and

$$f_2 = 1 - x_1 \tag{17}$$

are residual terms. In this case (unlike the least squares modification in Textbook Problem Formulation on page 192), the 2 problem formulations are equivalent and will have identical solutions.

Methods

In the `Dakota/test` directory, the `rosenbrock` executable (compiled from `rosenbrock.C`) returns an objective function as computed from Eq. (14) for use with optimization methods. The `rosenbrock_ls` executable (compiled from `rosenbrock_ls.C`) returns two least squares terms as computed from Eqs. (16) and (17) for use with least squares iterators. Both executables return analytic gradients of the function set (gradient of the objective function in `rosenbrock`, gradients of the least squares residuals in `rosenbrock_ls`) with respect to the design variables. The `dakota_rosenbrock.in` input file is used to solve both problems by toggling settings in the interface, responses, and method specifications. To run the optimization solution, select `'rosenbrock'` as the `analysis_driver` in the interface specification, select `num_objective_functions` to be 1 in the responses specification, and select an unconstrained or bound-constrained optimizer in the method specification (e.g., `dot_bfgs`, `optpp_bcq_newton`), e.g.:

```
interface,                                                              \
       application system,                                              \
         analysis_driver =    'rosenbrock'

variables,                                                              \
       continuous_design = 2                                            \
         cdv_initial_point    0.8      0.7                             \
         cdv_lower_bounds    -2.0     -2.0                             \
         cdv_upper_bounds     2.0      2.0                             \
         cdv_descriptor       'x1'     'x2'

responses,                                                              \
       num_objective_functions = 1                                      \
       analytic_gradients                                               \
       no_hessians

method,                                                                 \
       optpp_bcq_newton,                                                \
        max_iterations = 500                                            \
        convergence_tolerance = 1e-10
```

To run the least squares solution, select `'rosenbrock_ls'` as the `analysis_driver` in the interface specification, select `num_least_squares_terms` to be 2 in the responses specification, and select a Gauss-Newton iterator in the method specification (i.e., `optpp_g_newton` or `optpp_bcg_newton`), e.g.:

```
interface,                                                              \
       application system,                                              \
         analysis_driver =    'rosenbrock_ls'

variables,                                                              \
       continuous_design = 2                                            \
         cdv_initial_point    0.8      0.7                             \
         cdv_lower_bounds    -2.0     -2.0                             \
         cdv_upper_bounds     2.0      2.0                             \
         cdv_descriptor       'x1'     'x2'

responses,                                                              \
       num_least_squares_terms = 2                                      \
       analytic_gradients                                               \
       no_hessians

method,                                                                 \
       optpp_bcg_newton,                                                \
        max_iterations = 500                                            \
        convergence_tolerance = 1e-10
```

Results

The optimal solution, solved either as a least squares problem or an optimization problem, is:

$x_1 = 1.0$
$x_2 = 1.0$

with

$f^* = 0.0$

In comparing the two approaches, one would expect the Gauss-Newton approach to be more efficient since it exploits the special-structure of a least squares objective function. From a good initial guess, this expected behavior is observed. Starting from `cdv_initial_point = 0.8, 0.7`, the `optpp_bcg_newton` method converges in only 3 function and gradient evaluations while the `optpp_bcq_newton` method requires 14 function and gradient

evaluations to achieve similar accuracy. Starting from a poorer initial guess (e.g., `cdv_initial_point = -1.2, 1.0` as specified in `Dakota/test/dakota_rosenbrock.in`), the trend is less obvious since both methods spend several evaluations finding the vicinity of the minimum (total function and gradient evaluations = 24 for `optpp_bcq_newton` and 29 for `optpp_bcg_newton`). However, once the vicinity is located, convergence is much more rapid with the Gauss-Newton approach (11 orders of magnitude reduction in the objective function in 1 function and gradient evaluation) than with the quasi-Newton approach (12 orders of magnitude reduction in the objective function in 10 function and gradient evaluations).

Shown below is the DAKOTA output for the `optpp_bcg_newton` method starting from `cdv_initial_point = 0.8, 0.7`:

```
MPI initialized with 1 processors.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
no method_pointer: last specifications parsed will be used
methodName = optpp_bcg_newton
gradientType = analytic
hessianType = none
Running MPI executable in serial mode.
Running Single Method Strategy...

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      8.0000000000e-01 x1
                      7.0000000000e-01 x2

(rosenbrock_ls /var/tmp/aaaa000Sg /var/tmp/baaa000Sg)
Removing /var/tmp/aaaa000Sg and /var/tmp/baaa000Sg

Active response data for function evaluation 1:
Active set vector = { 3 3 }
                      6.0000000000e-01 least_sq_term1
                      2.0000000000e-01 least_sq_term2
 [ -1.6000000000e+01  1.0000000000e+01  ] least_sq_term1 gradient
 [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient


    nlf2_evaluator_gn results: objective fn. =
    4.0000000000e-01
    nlf2_evaluator_gn results: objective fn. gradient =
 [ -1.9600000000e+01  1.2000000000e+01 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  5.1400000000e+02 -3.2000000000e+02
   -3.2000000000e+02  2.0000000000e+02 ]]


------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                      9.9999528206e-01 x1
                      9.5999243139e-01 x2

(rosenbrock_ls /var/tmp/caaa000Sg /var/tmp/daaa000Sg)
Removing /var/tmp/caaa000Sg and /var/tmp/daaa000Sg

Active response data for function evaluation 2:
Active set vector = { 3 3 }
                     -3.9998132752e-01 least_sq_term1
                      4.7179400000e-06 least_sq_term2
 [ -1.9999905641e+01  1.0000000000e+01  ] least_sq_term1 gradient
```

```
 [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient


    nlf2_evaluator_gn results: objective fn. =
    1.5998506239e-01
    nlf2_evaluator_gn results: objective fn. gradient =
 [  1.5999168181e+01 -7.9996265504e+00 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  8.0199245130e+02 -3.9999811282e+02
   -3.9999811282e+02  2.0000000000e+02 ]]


------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                     9.9999904378e-01 x1
                     9.9999808275e-01 x2

(rosenbrock_ls /var/tmp/eaaa000Sg /var/tmp/faaa000Sg)
Removing /var/tmp/eaaa000Sg and /var/tmp/faaa000Sg

Active response data for function evaluation 3:
Active set vector = { 3 3 }
                    -4.8109144446e-08 least_sq_term1
                     9.5621999996e-07 least_sq_term2
 [ -1.9999980876e+01  1.0000000000e+01  ] least_sq_term1 gradient
 [ -1.0000000000e+00  0.0000000000e+00  ] least_sq_term2 gradient


    nlf2_evaluator_gn results: objective fn. =
    9.1667117810e-13
    nlf2_evaluator_gn results: objective fn. gradient =
 [  1.1923937841e-08 -9.6218288892e-07 ]
    nlf2_evaluator_gn results: objective fn. Hessian =
[[  8.0199847008e+02 -3.9999961752e+02
   -3.9999961752e+02  2.0000000000e+02 ]]


<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 3 total (3 new, 0 duplicate)
<<<<< Best design parameters  =
                     9.9999904378e-01 x1
                     9.9999808275e-01 x2
<<<<< Best objective function =
                     9.1667117810e-13
Run time from MPI_Init to MPI_Finalize is 7.8900400000e-01 seconds
```

Cylinder Head Example

Cylinder Head Problem Formulation

The cylinder head example problem arose as a simple demonstration problem for the Technologies Enabling Agile Manufacturing (TEAM) project. Its formulation is stated as

minimize

$$\mathtt{f} = -1\left(\frac{\mathtt{horsepower}}{250} + \frac{\mathtt{warranty}}{100000}\right) \tag{18}$$

subject to

$$g_1 = \sigma_{\mathtt{max}} - 0.5\sigma_{\mathtt{yield}} \leq 0 \tag{19}$$

$$g_2 = 100000 - \mathtt{warranty} \leq 0 \tag{20}$$

$$g_3 = \mathtt{time}_{\mathtt{cycle}} - 60 \leq 0 \tag{21}$$

$$1.5 \leq \mathtt{d}_{\mathtt{intake}} \leq 2.164 \tag{22}$$

$$0.0 \leq \mathtt{flatness} \leq 4.0 \tag{23}$$

This formulation seeks to simultaneously maximize normalized engine horsepower and engine warranty over variables of valve intake diameter ($\mathtt{d}_{\mathtt{intake}}$) in inches and overall head flatness ($\mathtt{flatness}$) in thousandths of an inch subject to constraints that the maximum stress cannot exceed half of yield, that warranty must be at least 100000 miles, and that manufacturing cycle time must be less than 60 seconds. The objective function and constraints are related analytically to the design variables according to the following simple expressions:

$$\mathtt{warranty} = 100000 + 15000(4 - \mathtt{flatness}) \tag{24}$$

$$\mathtt{time}_{\mathtt{cycle}} = 45 + 4.5(4 - \mathtt{flatness})^{1.5} \tag{25}$$

$$\mathtt{horsepower} = 250 + 200\left(\frac{\mathtt{d}_{\mathtt{intake}}}{1.8333} - 1\right) \tag{26}$$

$$\sigma_{\mathtt{max}} = 750 + \frac{1}{(\mathtt{t}_{\mathtt{wall}})^{2.5}} \tag{27}$$

$$\mathtt{t}_{\mathtt{wall}} = \mathtt{offset}_{\mathtt{intake}} - \mathtt{offset}_{\mathtt{exhaust}} - \frac{(\mathtt{d}_{\mathtt{intake}} + \mathtt{d}_{\mathtt{exhaust}})}{2} \tag{28}$$

where the constants in Eqns. (19) and (28) assume the following values: $\sigma_{yield} = 3000$, $offset_{intake} = 3.25$, $offset_{exhaust} = 1.34$, and $d_{exhaust} = 1.556$.

Methods

In the `Dakota/test` directory, the `dakota_cyl_head.in` input file is used to execute the cylinder head example. This file is shown below:

```
interface,                                                          \
       application system,                                          \
         asynchronous                                               \
         analysis_driver=  'cyl_head'

variables,                                                          \
       continuous_design = 2                                        \
         cdv_initial_point    1.8    1.0\
         cdv_upper_bounds     2.164  4.0\
         cdv_lower_bounds     1.5    0.0\
         cdv_descriptor 'intake_dia' 'flatness'

responses,                                                          \
       num_objective_functions = 1                                  \
       num_nonlinear_constraints = 3                                \
       numerical_gradients                                          \
         method_source dakota                                       \
         interval_type central                                      \
         fd_step_size = 1.e-4                                       \
       no_hessians

method,                                                             \
        npsol_sqp                                                   \
         convergence_tolerance = 1.e-8                              \
#        linear_constraints = 1. 1. -3.7\
         output verbose
```

The interface keyword specifies use of the `cyl_head` executable (compiled from `Dakota/test/cyl_head.C`) as the simulator. The variables and responses keywords specify the data sets to be used in the iteration by providing the initial point, descriptors, and upper and lower bounds for two continuous design variables and by specifying the presence of one objective function, three constraints, and analytic gradients in the problem. The method keyword specifies the use of the `npsol_sqp` method to solve this constrained optimization problem. No strategy keyword is specified, so the default `single_method` strategy is used.

Optimization Results

The solution for the constrained optimization problem is:

$intake\_dia = 2.122$
$flatness = 1.769$

with

$f^* = -2.461$
$g_1^* = 0.0$ (active)
$g_2^* = -0.3347$ (inactive)

g_2* = 0.0 (active)

which corresponds to the following optimal response quantities:

```
warranty = 133472
cycle_time = 60
horse_power = 281.579
max_stress = 1500
```

The DAKOTA output follows:

```
MPI initialized with 1 processors.
Writing new restart file dakota.rst
Constructing Single Method Strategy...
no method_pointer: last specifications parsed will be used
methodName = npsol_sqp
gradientType = analytic
hessianType = none

NPSOL option settings:
----------------------
Verify Level               = -1
Major Print Level          = 20
Function Precision         = 1e-10
Linesearch Tolerance       = 0.9
Major Iteration Limit      = 100
Optimality Tolerance       = 1e-08
NOTE: NPSOL's convergence tolerance is not a relative tolerance.
      See pp. 21-22 of NPSOL manual for description.
Derivative Level           = 3
Running MPI executable in serial mode.
Running Single Method Strategy...




                     NPSOL  ---  Version 4.06-2     Nov  1992
                     ========================================

------------------------------
Begin Function Evaluation    1
------------------------------
Parameters for function evaluation 1:
                      1.8000000000e+00 intake_dia
                      1.0000000000e+00 flatness

(cyl_head /var/tmp/aaaa0010M /var/tmp/baaa0010M)
In cyl_head evaluator:
warranty = 145000
cycle_time = 68.3827
wall_thickness = 0.232
horse_power = 246.399
max_stress = 788.573
Removing /var/tmp/aaaa0010M and /var/tmp/baaa0010M

Active response data for function evaluation 1:
Active set vector = { 3 3 3 3 }
                     -2.4355973813e+00 obj_fn
                     -4.7428486677e-01 nln_con1
                     -4.5000000000e-01 nln_con2
                      1.3971143170e-01 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  1.3855136438e-01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.9485571585e-01  ] nln_con3 gradient
```

```
  Maj  Mnr    Step  Fun  Merit function  Violtn Norm gZ   nZ Penalty Conv
    0    2 0.0E+00    1 -1.98878999E+00 3.9E-01 0.0E+00    0 4.6E+01 F TF

------------------------------
Begin Function Evaluation    2
------------------------------
Parameters for function evaluation 2:
                     2.1640000000e+00 intake_dia
                     1.7169994018e+00 flatness

(cyl_head /var/tmp/caaa0010M /var/tmp/daaa0010M)
In cyl_head evaluator:
warranty = 134245
cycle_time = 60.5229
wall_thickness = 0.05
horse_power = 286.116
max_stress = 2538.85
Removing /var/tmp/caaa0010M and /var/tmp/daaa0010M

Active response data for function evaluation 2:
Active set vector = { 3 3 3 3 }
                    -2.4869127193e+00 obj_fn
                     6.9256958800e-01 nln_con1
                    -3.4245008973e-01 nln_con2
                     8.7142207939e-03 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  2.9814239700e+01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6998301774e-01  ] nln_con3 gradient


    1    1 1.0E+00    2 -2.46707673E+00 6.9E-01 0.0E+00    0 6.8E+00 F TF

------------------------------
Begin Function Evaluation    3
------------------------------
Parameters for function evaluation 3:
                     2.1407705098e+00 intake_dia
                     1.7682646453e+00 flatness

(cyl_head /var/tmp/eaaa0010M /var/tmp/faaa0010M)
In cyl_head evaluator:
warranty = 133476
cycle_time = 60.0029
wall_thickness = 0.0616147
horse_power = 283.581
max_stress = 1811.18
Removing /var/tmp/eaaa0010M and /var/tmp/faaa0010M

Active response data for function evaluation 3:
Active set vector = { 3 3 3 3 }
                    -2.4690845846e+00 obj_fn
                     2.0745219855e-01 nln_con1
                    -3.3476030320e-01 nln_con2
                     4.9104542814e-05 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  1.4352331520e+01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6806368014e-01  ] nln_con3 gradient



------------------------------
Begin Function Evaluation    4
------------------------------
Parameters for function evaluation 4:
                     2.1607040498e+00 intake_dia
                     1.7242732458e+00 flatness

(cyl_head /var/tmp/gaaa0010M /var/tmp/haaa0010M)
In cyl_head evaluator:
warranty = 134136
```

```
cycle_time = 60.4487
wall_thickness = 0.051648
horse_power = 285.756
max_stress = 2399.55
Removing /var/tmp/gaaa0010M and /var/tmp/haaa0010M

Active response data for function evaluation 4:
Active set vector = { 3 3 3 3 }
                     -2.4843831483e+00 obj_fn
                      5.9970320968e-01 nln_con1
                     -3.4135901313e-01 nln_con2
                      7.4787762078e-03 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  2.6615351511e+01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6971201116e-01  ] nln_con3 gradient


    2    0 1.4E-01    4 -2.46179789E+00 6.0E-01 0.0E+00    0 6.8E+00 F TF

------------------------------
Begin Function Evaluation    5
------------------------------
Parameters for function evaluation 5:
                      2.1381718203e+00 intake_dia
                      1.7683406996e+00 flatness

(cyl_head /var/tmp/iaaa0010M /var/tmp/jaaa0010M)
In cyl_head evaluator:
warranty = 133475
cycle_time = 60.0022
wall_thickness = 0.0629141
horse_power = 283.298
max_stress = 1757.23
Removing /var/tmp/iaaa0010M and /var/tmp/jaaa0010M

Active response data for function evaluation 5:
Active set vector = { 3 3 3 3 }
                     -2.4679389967e+00 obj_fn
                      1.7148906598e-01 nln_con1
                     -3.3474889506e-01 nln_con2
                      3.6322686164e-05 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  1.3341388781e+01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6806081643e-01  ] nln_con3 gradient



------------------------------
Begin Function Evaluation    6
------------------------------
Parameters for function evaluation 6:
                      2.1523079940e+00 intake_dia
                      1.7406938490e+00 flatness

(cyl_head /var/tmp/kaaa0010M /var/tmp/laaa0010M)
In cyl_head evaluator:
warranty = 133890
cycle_time = 60.2818
wall_thickness = 0.055846
horse_power = 284.84
max_stress = 2106.81
Removing /var/tmp/kaaa0010M and /var/tmp/laaa0010M

Active response data for function evaluation 6:
Active set vector = { 3 3 3 3 }
                     -2.4782556582e+00 obj_fn
                      4.0454151196e-01 nln_con1
                     -3.3889592265e-01 nln_con2
                      4.6970356970e-03 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
```

```
 [  2.0246335086e+01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6909862055e-01  ] nln_con3 gradient


    3    0 3.7E-01    6 -2.45857429E+00 4.0E-01 0.0E+00    0 6.8E+00 F TF

------------------------------
Begin Function Evaluation    7
------------------------------
Parameters for function evaluation 7:
                      2.1323270192e+00 intake_dia
                      1.7684707504e+00 flatness

(cyl_head /var/tmp/maaa0010M /var/tmp/naaa0010M)
In cyl_head evaluator:
warranty = 133473
cycle_time = 60.0009
wall_thickness = 0.0658365
horse_power = 282.66
max_stress = 1649.15
Removing /var/tmp/maaa0010M and /var/tmp/naaa0010M

Active response data for function evaluation 7:
Active set vector = { 3 3 3 3 }
                     -2.4653685666e+00 obj_fn
                      9.9434951763e-02 nln_con1
                     -3.3472938744e-01 nln_con2
                      1.4466560964e-05 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  1.1381130512e+01  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6805591946e-01  ] nln_con3 gradient


    4    0 1.0E+00    7 -2.46038884E+00 9.9E-02 0.0E+00    0 6.8E+00 F TF

------------------------------
Begin Function Evaluation    8
------------------------------
Parameters for function evaluation 8:
                      2.1235901936e+00 intake_dia
                      1.7685568322e+00 flatness

(cyl_head /var/tmp/oaaa0010M /var/tmp/paaa0010M)
In cyl_head evaluator:
warranty = 133472
cycle_time = 60
wall_thickness = 0.0702049
horse_power = 281.707
max_stress = 1515.74
Removing /var/tmp/oaaa0010M and /var/tmp/paaa0010M

Active response data for function evaluation 8:
Active set vector = { 3 3 3 3 }
                     -2.4615425280e+00 obj_fn
                      1.0493396662e-02 nln_con1
                     -3.3471647517e-01 nln_con2
                      1.4443046759e-10 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  9.0893472783e+00  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6805267803e-01  ] nln_con3 gradient



  Maj  Mnr    Step  Fun  Merit function  Violtn Norm gZ   nZ Penalty Conv
    5    0 1.0E+00    8 -2.46101307E+00 1.0E-02 0.0E+00    0 6.8E+00 F TF

------------------------------
Begin Function Evaluation    9
```

```
------------------------------
Parameters for function evaluation 9:
                     2.1224357217e+00 intake_dia
                     1.7685568330e+00 flatness

(cyl_head /var/tmp/qaaa0010M /var/tmp/raaa0010M)
In cyl_head evaluator:
warranty = 133472
cycle_time = 60
wall_thickness = 0.0707821
horse_power = 281.581
max_stress = 1500.22
Removing /var/tmp/qaaa0010M and /var/tmp/raaa0010M

Active response data for function evaluation 9:
Active set vector = { 3 3 3 3 }
                    -2.4610386667e+00 obj_fn
                     1.4914647635e-04 nln_con1
                    -3.3471647505e-01 nln_con2
                     9.9882324633e-12 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  8.8325450545e+00  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6805267800e-01  ] nln_con3 gradient


    6    0 1.0E+00    9 -2.46103129E+00 1.5E-04 0.0E+00    0 6.8E+00 F TF

------------------------------
Begin Function Evaluation   10
------------------------------
Parameters for function evaluation 10:
                     2.1224188357e+00 intake_dia
                     1.7685568331e+00 flatness

(cyl_head /var/tmp/saaa0010M /var/tmp/taaa0010M)
In cyl_head evaluator:
warranty = 133472
cycle_time = 60
wall_thickness = 0.0707906
horse_power = 281.579
max_stress = 1500
Removing /var/tmp/saaa0010M and /var/tmp/taaa0010M

Active response data for function evaluation 10:
Active set vector = { 3 3 3 3 }
                    -2.4610312969e+00 obj_fn
                     3.1248197141e-08 nln_con1
                    -3.3471647503e-01 nln_con2
                    -6.8171024381e-12 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  8.8288585865e+00  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6805267799e-01  ] nln_con3 gradient


    7    0 1.0E+00   10 -2.46103130E+00 3.1E-08 0.0E+00    0 6.8E+00 T TF

------------------------------
Begin Function Evaluation   11
------------------------------
Parameters for function evaluation 11:
                     2.1224188321e+00 intake_dia
                     1.7685568331e+00 flatness

(cyl_head /var/tmp/uaaa0010M /var/tmp/vaaa0010M)
In cyl_head evaluator:
warranty = 133472
cycle_time = 60
wall_thickness = 0.0707906
horse_power = 281.579
max_stress = 1500
```

```
Removing /var/tmp/uaaa0010M and /var/tmp/vaaa0010M

Active response data for function evaluation 11:
Active set vector = { 3 3 3 3 }
                     -2.4610312954e+00 obj_fn
                     -5.3569115810e-10 nln_con1
                     -3.3471647503e-01 nln_con2
                     -6.8171024381e-12 nln_con3
 [ -4.3644298963e-01  1.5000000000e-01  ] obj_fn gradient
 [  8.8288578008e+00  0.0000000000e+00  ] nln_con1 gradient
 [  0.0000000000e+00  1.5000000000e-01  ] nln_con2 gradient
 [  0.0000000000e+00 -1.6805267799e-01  ] nln_con3 gradient


    8    0 1.0E+00   11 -2.46103130E+00 5.4E-10 0.0E+00    0 6.8E+00 T TT

 Exit NPSOL - Optimal solution found.

 Final nonlinear objective value =   -2.461031

NPSOL exits with INFORM code = 0 (see p. 8 of NPSOL manual)

NOTE: see Fortran device 9 file (fort.9 or ftn09)
      for complete NPSOL iteration history.

<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 11 total (11 new, 0 duplicate)
<<<<< Best design parameters  =
                      2.1224188321e+00 intake_dia
                      1.7685568331e+00 flatness
<<<<< Best objective function =
                     -2.4610312954e+00
<<<<< Best constraint values  =
                     -5.3569115810e-10
                     -3.3471647503e-01
                     -6.8171024381e-12
Run time from MPI_Init to MPI_Finalize is 1.6473130000e+00 seconds
```

Engineering Applications

Transportation Cask Example

In this example, use is made of C-shell scripting to coordinate pre-processing, invocation of analyses, and post-processing.

Work in progress

Alternate with workdir tagging: radar load spreader plate

GOMA/EXODUS Application Example

This tutorial is designed to give an experienced GOMA/EXODUS jockey an introduction into tying the DAKOTA iterator toolkit to the GOMA simulation code. In addition to understanding GOMA and the EXODUS file format, the user is assumed to have an understanding of a programming language such as C or FORTRAN. Although many of the examples will be presented in C, the programs can just as easily be written in FORTRAN.

Standard text_book example

Problem formulation:

The problem to be solved in this portion of the tutorial is the text_book example:

$$f = (x_1 - 1)^4 + (x_2 - 1)^4 \tag{29}$$

$$g_1 = x_1^2 - \frac{x_2}{2} \leq 0$$

$$g_2 = x_2^2 - 0.5 \leq 0$$

subject to simple bounds on the variables: x_1 and x_2 range between -10 and +10.

Dakota_sample.in problem description file:

Sections are delimited by newline characters. Therefore, to continue a section onto multiple lines, the back-slash character is needed to escape the newline. Input is order-independent and white-space insensitive. Keywords may be abbreviated so long as the abbreviation is unique. Comments are preceded by #.The definitive resource for input grammar is `Dakota/src/dakota.input.spec`.

```
# DAKOTA INPUT FILE - dakota_textbook.in
# Interface section specification
```

```
# NOTES: Interfaces are 1 of 3 main types: application interfaces are used
#      for interfacing with simulation codes, approximation interfaces use
#     inexpensive design space approximations in place of expensive
#     simulations, and test interfaces use linked-in test functions for
#     algorithm testing purposes (to eliminate system call overhead).
#     Application interfaces can be further categorized into system and
#     direct types. The system type uses system calls to invoke the
#     simulation, while the direct type uses the same constructs as the test
#     interface for linked-in simulation codes.  Both application interface
#     types use analysis_driver, input_filter, and output_filter
#     specifications. The system type additionally uses parameters_file,
#     results_file, analysis_usage, file_tag, and file_save specifications.
#     The analysis_driver provides the name of the analysis executable,
#     driver script, or linked module; the input_filter and output_filter
#      provide pre- and post-processing for the analysis in the procedure of
#     mapping parameters into responses (default = NO_FILTER); the
#     parameters_file and results_file are data files which Dakota creates
#     and reads, respectively, in the system call case (default = Unix temp
#     files); analysis_usage defines nontrivial command syntax (default =
#     standard syntax), file_tag controls the unique tagging of data files
#     with function evaluation number (default = no tagging), and file_save
#     controls whether or not file cleanup operations are performed (default
#     = data files are removed when no longer in use). Most settings are
#     optional with meaningful defaults as shown above. Refer to the
#     Interface Commands section in the User's Instructions manual for
#     additional information.

interface,\
       application system,\
         input_filter    =       'NO_FILTER'\
         output_filter   =       'NO_FILTER'\
         analysis_driver =       'text_book'\
         parameters_file =       'text_book.in'\
         results_file    =       'text_book.out'\
         analysis_usage  =       'DEFAULT'\
         file_tag\
         file_save

# Variables specification
# NOTES:A variables set can contain design, uncertain, and state variables.
#      Design variables are those variables which an optimizer adjusts in
#      order to locate an optimal design.  Each of the n design parameters
#      can have an initial point, a lower bound, an upper bound, and a
#       descriptive tag.  Uncertain variables are those variables which are
#      characterized by probability distributions.  Each uncertain variable
#        specification can contain a distribution type, a mean, a standard
#        deviation, a lower bound, an upper bound, a histogram filename and a
#        descriptive tag.  State variables are "other" variables which are to
#        be mapped through the interface.  Each state variable specification
#      can have an initial state and a descriptor.  State variables provide a
#    convenience mechanism for parameterizing additional model inputs, such
#      as mesh density, solver convergence tolerance and time step controls,
#        and will be used to enact model adaptivity in future strategy
#        developments.

variables,\
       continuous_design = 2\
         cdv_initial_point    0.9    1.1\
         cdv_upper_bounds     5.8    2.9\
         cdv_lower_bounds     0.5   -2.9\
         cdv_descriptor       'x1'   'x2'

# Responses specification
# NOTES: This specification implements a generalized Dakota data set by
#       specifying a set of functions and the types of gradients and hessians
#      for these functions.  Optimization data sets require specification of
#      num_objective_functions, num_linear_constraints, and
#      num_nonlinear_constraints. Multiobjective opt. is not yet supported,
#      so num_objective_functions must be = 1. Uncertainty quantification
#      data sets are specified by num_response_functions.  Nonlinear least
#     squares data sets are specified with num_least_squares_terms.
#      Gradient type specification may be no_gradients, analytic_gradients,
```

```
#     numerical_gradients or mixed_gradients:
#        > no_gradients is invalid with gradient-based opt. methods
#        > no_gradients or analytic_gradients are complete specifications
#        > if numerical_gradients, then:
#             >> method_source = vendor OR dakota
#             >> interval_type = forward OR cental
#             >> fd_step_size = <float>
#          are additional optional parameters in the specification.
#        > mixed_gradients uses id_numerical & id_analytic lists to specify
#         the gradient types for different function numbers.  This capability
#        is not yet completely implemented within the Iterators.
#        Hessian type specification may currently be no_hessians or
#        analytic_hessians.  The only optimizers to currently support
#        analytic_hessians are the OPT++ full Newton methods.

responses,\
       num_objective_functions = 1\
       num_linear_constraints = 0\
       num_nonlinear_constraints = 2\
       analytic_gradients\
       no_hessians

# Strategy specification
# NOTES: Contains specifications for hybrid, SAO, and OUU strategies.  The
#     single_method strategy is a “fall through” strategy, in that in only
#     invokes a single iterator.  If no strategy specification appears, then
#     single_method is the default.

strategy,\
       single_method

# Method specification
# NOTES: method can currently be dot_frcg, dot_mmfd, dot_bfgs, dot_slp,
#        dot_sqp, npsol_sqp, optpp_cg, optpp_q_newton, optpp_g_newton,
#        optpp_newton, optpp_fd_newton, optpp_ba_newton, optpp_baq_newton,
#        optpp_bc_newton, optpp_bcq_newton, optpp_bc_ellipsoid, optpp_pds,
#        optpp_test_new, sgopt_pga_real, sgopt_pga_int, sgopt_coord_ps,
#        sgopt_coord_sps, sgopt_solis_wets, sgopt_strat_mc, nond_probability,
#     nond_mean_value, or parameter_study. Most method control parameters
#     are optional with meaningful defaults, although sgopt_coord_ps,
#     sgopt_coord_sps, parameter_study, nond_probability, and
#     nond_mean_value have some required control parameters. Default values
#     for optional parameters are defined in the DataMethod class
#     constructor and are documented in the Method Commands section of the
#        User’s Instructions manual.

method,\
         dot_sqp,\
          max_iterations = 50,\
          convergence_tolerance = 1e-4\
          output verbose\
          optimization_type minimize
```

Simulator file text_book.C:

This simple application program reads the parameters and writes the responses directly; therefore, the NO_FILTER option is be used. The output must be formatted based on the DakotaResponse IO operators.

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <rw/cstring.h>

#ifdef SYMANTEC
#include <console.h>
#endif
```

```
double eval(const double* x, int len);
int main(int argc, char** argv)
{
#ifdef SYMANTEC
  argc = ccommand(&argv);

  for(int num=0; num<argc; num++) {
    cout << argv[num] << " ";
  }
  cout << '\n';
#endif

  ifstream fin(argv[1]);
  ofstream fout(argv[2]);

  // Get the first line and use info for array allocation
  int num_vars, num_fns;
  RWCString vars_text, fns_text;
  fin >> num_vars >> vars_text >> num_fns >> fns_text;

  // Get the parameter vector and ignore the labels
  //vector<double> x(num_vars);
  double* x = new double [num_vars];
  int i;
  for(i=0; i<num_vars; i++) {
    fin >> x[i];
    fin.ignore(256, '\n');
  }

  // Get the ASV vector and ignore the labels
  int* ASV = new int [num_fns];
  for(i=0; i<num_fns; i++) {
    fin >> ASV[i];
    fin.ignore(256, '\n');
  }

  // Compute the results and output them directly to argv[2] (the NO_FILTER
  // option is used).  Response tags are now optional; output them for ease
  // of results readability.
  fout.precision(10);
  fout.setf(ios::scientific);
  fout.setf(ios::right);
  // **** f:
  if (ASV[0]==1 || ASV[0]==3 || ASV[0]==5 || ASV[0]==7)
    fout << "                     " << eval(x, num_vars) << " f\n";

  // **** c1:
  if (num_fns>1) {
    if (ASV[1]==1 || ASV[1]==3 || ASV[1]==5 || ASV[1]==7)
      fout << "                     " << (x[0]*x[0] - 0.5*x[1]) << " c1\n";
  }

  // **** c2:
  if (num_fns>2) {
    if (ASV[2]==1 || ASV[2]==3 || ASV[2]==5 || ASV[2]==7)
      fout << "                     " << (x[1]*x[1] - 0.5     ) << " c2\n";
  }

  // **** df/dx:
  if (ASV[0]==2 || ASV[0]==3 || ASV[0]==6 || ASV[0]==7) {
    fout << "[ ";
    for (i=0; i<num_vars; i++)
      fout << 4.*pow(x[i]-1,3) << " ";
    fout << "]\n";
  }

  // **** dc1/dx:
  if (num_fns>1) {
    if (ASV[1]==2 || ASV[1]==3 || ASV[1]==6 || ASV[1]==7) {
      fout << "[ " << 2.*x[0] << " " << -0.5;
      for (i=3; i<=num_vars; i++)
        fout << " " << 0.0;
```

```
      fout << " ]\n";
    }
  }

  // **** dc2/dx:
  if (num_fns>2) {
    if (ASV[2]==2 || ASV[2]==3 || ASV[2]==6 || ASV[2]==7) {
      fout << "[ " << 0.0 << " " << 2.*x[1];
      for (i=3; i<=num_vars; i++)
        fout << " " << 0.0;
      fout << " ]\n";
    }
  }

  // **** d^2f/dx^2: (full Newton unconstrained opt.)
  if (ASV[0]>=4) {
    fout << "[[ ";
    for (i=0; i<num_vars; i++)
      for (int j=0; j<num_vars; j++)
        if (i==j)
          fout << 12.*pow(x[i]-1,2) << " ";
        else
          fout << 0. << " ";
    fout << "]]\n";
  }

  // **** d^2c1/dx^2: (ParamStudy testing of multiple Hessian matrices)
  if (num_fns>1) {
    if (ASV[1]>=4) {
      fout << "[[ ";
      for (i=0; i<num_vars; i++)
        for (int j=0; j<num_vars; j++)
          if (i==0 && j==0)
            fout << 2. << " ";
          else
            fout << 0. << " ";
      fout << "]]\n";
    }
  }

  // **** d^2c2/dx^2: (ParamStudy testing of multiple Hessian matrices)
  if (num_fns>2) {
    if (ASV[2]>=4) {
      fout << "[[ ";
      for (i=0; i<num_vars; i++)
        for (int j=0; j<num_vars; j++)
          if (i==1 && j == 1)
            fout << 2. << " ";
          else
            fout << 0. << " ";
      fout << "]]\n";
    }
  }

  fout << flush;
  delete [] x;
  delete [] ASV;

  return 0;
}

//double eval(const vector<double>& x)
double eval(const double* x, int len)
{
  double value=0;

  for(int i=len; i--; ) {
    value += pow(x[i]-1, 4);
  }

  return value;
}
```

Invokation of text_book:

The command syntax which DAKOTA will use is as shown below. Parameters and results file names will be passed on the command line to the specified executable and file tagging will be employed to keep the file names unique. The names of the parameters and results files are passed on the command line for the convenience of the application developer, since these arguments can be used to remove hard coding of file names and improve generality:

```
text_book text_book.in.1 text_book.out.1
```

The text_book.in.1 parameters file is:

```
 2 variables 3 functions
 9.0000000000e-01 x1
 1.1000000000e+00 x2
                1 ASV_1
                1 ASV_2
                1 ASV_3
```

and the text_book.out.1 results files is:

```
2.0000000000e-04 f
2.6000000000e-01 c1
7.1000000000e-01 c2
```

Results:

The dot_sqp method converges to the optimal solution in 17 total function evaluations when foward finite differences are used

```
<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 17 total (16 new, 1 duplicate)
<<<<< Best design parameters  =
                     5.9442052455e-01 x1
                     7.0668310706e-01 x2
<<<<< Best objective function =
                     3.4460496673e-02
<<<<< Best constraint values  =
                    -5.7935237028e-06
                    -5.9898619602e-04
```

Example text_book recast in GOMA format: Filter Introduction

There are several ways of interfacing DAKOTA with a simulation code. The method used here applies DAKOTA's 1-piece Interface capability. For this method DAKOTA makes one system call per function evaluation and all control over the evaluation is given to the user. DAKOTA also has a 3-piece Interface capability which performs separate system calls for the input filter, simulation code, and the output filter, in that order, to evaluation of the cost function and constraints. In the optimization problems described here, the evaluation of the cost function is

performed by a combination of C programs and a supervisory UNIX shell program using the 1-piece Interface capability.

Figure 1 outlines how variables and response data are passed as files and how the different codes interact. A DAKOTA input file (e.g., `dakota_sample.in`) specifies control parameters for the DAKOTA optimization run such as names of the variables and response file `params.in` and `results.out`, respectively, the number of design variables, their bounds and initial values, information concerning the number of constraints and how gradients are calculated, and the optimization method desired.

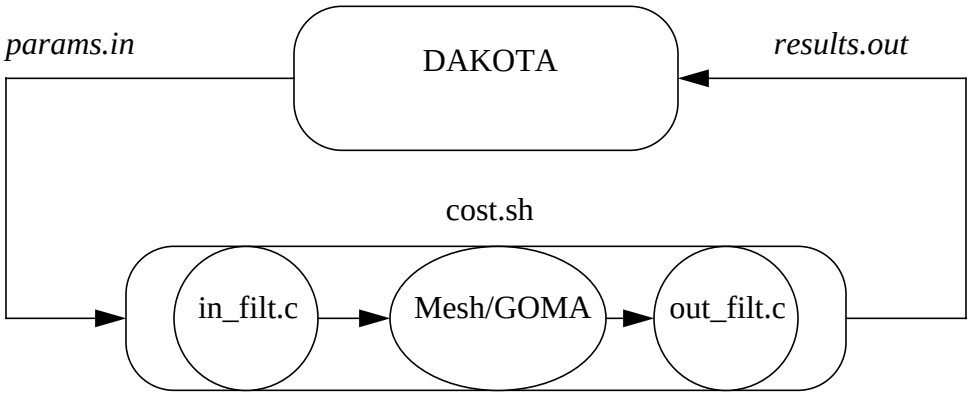


Figure 1. DAKOTA interface scheme

Just prior to requesting a function evaluation, DAKOTA writes the file `params.in`. This file contains the current values of the design variables and an **active set vector** code request for values of the function and constraints, their gradients, and/or the Hessian matrix. DAKOTA then spawns a system call and `params.in` is read by the user's shell program `cost.sh`. The shell program is fairly simple in that all it does is execute `in_filt.c`, GOMA, and `out_filt.c` in the appropriate order.

The input filter program, `in_filt.c`, places design variables identified in file `params.in` into a file that is formatted for use by APREPRO. This file is "included" into the mesh generator file or into the GOMA input deck. GOMA is then run and an EXODUS file is generated. The output filter program, `out_filt.c`, then reads the EXODUS file, extracts the necessary results, computes the cost function and the value of the constraints, and then writes the file `results.out` in DAKOTA readable format.

The programs `in_filt.c` and `out_filt.c` are written in a general format. The input filter can be run without any modification in most optimizations. A skeleton output filter is provided that only requires the subroutines to evaluate the cost functions and the constraints. The code for writing the file `results.out` file is also provided.

DAKOTA Filter Tutorial

The `text_book` example will be revisited in this portion of the tutorial, and will be recast in the form used by the GOMA applications. The problem formulation, as before, is

$$f = (x_1 - 1)^4 + (x_2 - 1)^4 \tag{30}$$

$$g_1 = x_1^2 - \frac{x_2}{2} \leq 0$$

$$g_2 = x_2^2 - 0.5 \leq 0$$

subject to simple bounds on the variables: x_1 and x_2 which range between -10 and +10. The following steps are used to generate a 3-piece interface.

1. Change directories into the “tutorial” directory (this location will depend on the course you are taking and how you installed your software).

2. You will notice the directory contains five files: `in_filt.c`, `out_filt.c`, `dak_goma.h`, `cost.sh`, and `tut.in`. The file `tut.in` is the DAKOTA input specification as discussed earlier. Issue the following commands:

   ```
   more tut.in
   ```

   The first part of the file defines how the DAKOTA `interface` is set up. A slash, at the end of a line signifies a continuation. It must be present immediately before any carriage return prior to the end of a keyword specification (e.g. `interface`, `method`, `variables`,...) This syntax is necessary because the parser detects keyword input completion with a newline, so newlines entered for readability must be escaped with a ‘\’. Note that the communication files are set up to be named `params.in` and `results.out`. The cost function is evaluated in the file `cost.sh`, which is called whenever DAKOTA issues the command

   ```
   cost.sh params.in results.out
   ```

   to the operating system. The shell function `cost.sh` must therefore be coded with this in mind (as we will see next).

   ```
   interface,                                                    \
    application system,                                          \
     input_filter =          ‘NO_FILTER’                         \
     output_filter =         ‘NO_FILTER’                         \
     analysis_driver=        ‘cost.sh’                           \
     parameters_file=        ‘params.in’                         \
     results_file=           ‘results.out’                       \
     analysis_usage =        ‘DEFAULT’
   ```

   Next, the design variables are set up. Note that there are two design variables, x_1 and x_2, and the starting point is (2, 2). Each variable may range between -10 and +10.

   ```
   variables,                                                    \
    continuous_design = 2                                        \
   ```

```
  cdv_initial_point        2.0      2.0\
  cdv_upper_bounds     10.0     10.0\
  cdv_lower_bounds    -10.0    -10.0\
  cdv_descriptor       'x1'     'x2'
```

The response specification describes the number of constraints and the source of the gradients. In this problem and in the problems utilizing GOMA, the gradients are calculated using a forward difference scheme:

```
responses,                                                          \
 num_objective_functions = 1                                        \
 num_linear_constraints = 0                                         \
 num_nonlinear_constraints = 2                                      \
 numerical_gradients                                                \
       method_source  vendor           \
       interval_type forward           \
       fd_step_size = 0.001            \
 no_hessians
```

The last portion selects the optimization technique to be used.

```
method,                                          \
 dot_sqp,                                   \
       max_iterations = 50,                  \
       convergence_tolerance = 1e-8          \
       output verbose                        \
       optimization_type minimize
```

3. Now execute the command:

```
more cost.sh
```

This file is the supervisory file that controls the cost function evaluation. This simple example has no GOMA evaluation.

```
#! /bin/csh -f
#
#  This shell file evaluates the cost function
#    for a dakota run
#
in_filt $argv[1] out.app

## GOMA run goes here!!

out_filt $argv[1] $argv[2]
```

The input filter, `in_filt.c`, places the design variables identified in file `params.in` into a file, `out.app`. The file `out_filt.c` will take the file `out.app` and evaluate the cost function then write the file `results.out`. The first line of the file `cost.sh` is necessary for the shell to execute correctly. The variables `$argv[1]` and `$argv[2]` refer to the first argument and the second argument in the call statement.

4. Now look at the input filter using the command:

```
more in_filt.c
```

The first portion of the file sets up various definitions and prototypes the functions used in the program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

void input_filter(FILE *input_file, FILE *param_file);
```

The program is controlled from `main()`. This routine performs error checking on the number of arguments used to call the program, opens `params.in` for reading and `out.app` for writing and calls the subroutine to perform the filtering operation, `input_filt()`.

```
void main(int argc,char *argv[])
 {

 FILE *input_file, *param_file;

 if (argc<2)
   {
   printf(ÒNeed an output filename, exiting\nÓ);
   exit(-1);
   }

 if ((input_file=fopen(argv[1],ÓrÓ)) == NULL)
   {
   printf(ÒCouldnÕt open file: %s   exiting.\nÓ,argv[1]);
   exit(-1);
   }

 param_file=fopen(argv[2],ÓwÓ);

 input_filter(input_file, param_file);
 exit(0);



 }
```

The first line of `params.in` specifies the number of design variables (`n_param`) and a string (`tag`). The next `n_param` lines are the value of each of the design variables with an identification tag. The last lines are the ASV for the function and each of the constraints. The ASV can be ignored in this input filter since only function values will be returned. The initial `params.in` file for this problem is listed below:

```
2 variables 3 functions
2.0000000000e+00 x1
2.0000000000e+00 x2
                1 ASV_1
                1 ASV_2
                1 ASV_3
```

The last portion of the file `in_filt.c` is the input filter subroutine. It just reads `params.in` and writes `out.app` using the format in the above paragraph.

```
void input_filter(FILE *input_file, FILE *param_file)
{
int i, n_param,n_g;
float dum_param;
char tag1[10],tag2[10];

fscanf(input_file,"%d %s %d %s",&n_param,tag1,&n_g,tag2);

for (i=0;i<n_param;i++)
 {
 fscanf(input_file,"%f %s",&dum_param,tag1);
 fprintf(param_file,"#{%s = %f}\n",tag1,dum_param);
 }
}
```

The contents of the file `out.app` will look something like:

```
#{x1 = 2.000000}
#{x2 = 2.000000}
```

You will recognize this as input for APREPRO.

5. The final file is the output filter (`out_filt.c`). Normally, it reads an EXODUS file to get the results of a GOMA run. In this case, the file `out.app` will represent the EXODUS file to simplify the description of the filters. As with `in_filt.c`, the first lines set up definitions and prototypes for the remaining code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define NUM_PARAM 2
#define MAX_LINE 80

float cost_fun(FILE *exoid);

float *asv_read(FILE *input_file,int *n_param, int *n_g, int **asv);

void output_filter(int *asv, int n_param, float *params, int n_g, FILE
*output_file);
```

The `main()` routine in `out_filt.c` once again controls the filter process as with `in_filt.c`. First some error checking is performed to insure the correct number of arguments are being passed. Next the files `params.in` and results.out are opened. The remaining functions are the meat of the program and will be discussed next.

```
int main(int argc,char *argv[])
{

int *asv, n_param,i, n_g;
float *params;
FILE *input_file, *output_file;

if (argc<3)
{
 printf("Need both input and output files specified, exiting \n");
 exit(-1);
}

input_file=fopen(argv[1],"r");

output_file=fopen(argv[2],"w");

params=asv_read(input_file,&n_param,&n_g,&asv);

output_filter(asv, n_param, params, n_g, output_file);

exit(0);
}
```

The subroutine `asv_read()` reads the `params.in` file returning the ASV information and the values of the parameters. This allows the program to correctly determine what DAKOTA is requesting and to allow the parameters to be available for the cost function and the constraint evaluation. The array `asv[]` and the array `params[]` are alloced in `asv_read()`. This is done here with the `calloc()` statement.

```
float *asv_read(FILE *input_file, int *n_param, int n_g, int **asv)
{
 int i;
 char junk1[MAX_LINE],junk2[MAX_LINE];
 float *params;

 fscanf(input_file,Ó%d %sÓ,n_param,junk1,n_g,junk2);
 *n_g = *n_g - 1;
```

```
 params= (float *)calloc(*n_param, sizeof(float));
 *avs=(int *)calloc(*n_g +1, sizeof(int));

 for (i=0;i<*n_param;i++) fscanf(input_file,Ó%f %s\nÓ,&params[i], junk);

 for (i=0;i<=*n_g;i++)
   {
   fscanf(input_file,Ó%d %s\nÓ,&(*asv)[i],junk1);
   }
 fclose(input_file);
 return(params);
}
```

The next subroutine is the actual output filter (`out_filt.c`). This subroutine opens the EXODUS file (in this case `out.app`) and evaluates the constraints, `g[n_g]` and the cost function, `J_cost`. In this example the cost function is evaluated using the routine `cost()` and the constraints are just combinations of the parameters. The remaining code preforms error checks on the ASV to be sure that the DAKOTA input specification is correct as far as the gradients and Hessians that can be provided by the output filter. It also writes out the results.out file with the appropriate information.

```
void output_filter(int *asv, int n_param, float *params, int n_g, FILE
 *output_file)
{
 int i;
 float J_cost;
 float *g;
 FILE *exoid;

 g=(float *)calloc(n_g ,sizeof(float));

 exoid=fopen(Òout.appÓ,ÓrÓ);
/* determine cost function and constraints*/

 g[0] =  params[0]*params[0] - params[1]/2.;
 g[1] = params[1]*params[1] -0.5;

 J_cost =  cost_fun(exoid);

 /* write dakota output file */

 if (asv[0]>3)
   {
   printf(ÒHessian is not available, exiting\nÓ);
   exit(-1);
   }

 if (asv[0]>2)
   {
   printf(ÒGradient is not available, exiting\nÓ);
   exit(-1);
   }

 if (asv[0]==1 || asv[0]==3 ||asv[0]==5)
   {
   fprintf(output_file,Ó%g f\nÓ,J_cost);
   }

 for (i=1;i<=n_g;i++)
   {
   if (asv[i]==1 || asv[i]==3 || asv[i]==5)
     fprintf(output_file,Ó%g c%d\nÓ,g[i-1], i);
   else
     {
     printf(ÒNumber of parameters is probably wrong:      exiting.\nÓ);
     exit(-1);
     }
   }
```

```
 free(g);
}
```

The final routine evaluates the cost function. In this case, this routine is exceptionally simple. It just reads the file out.app and runs the parameters through a formula:

```
float cost_fun(FILE *exoid)
{
int i;
float x[NUM_PARAM], J_cost, a, b;
char cdum[2];

for (i=0;i<NUM_PARAM;i++)
 {
 fscanf(exoid,"#{%s = %f}\n",cdum,&x[i]);
 printf(" x[%d] = %g \n",i,x[i]);
 }

a=(x[0]-1.);
b=(x[1]-1.);

J_cost = a*a*a*a + b*b*b*b;

return J_cost;
}
```

6. To compile a program with EXODUS subroutines in it, excute a command similar to:

```
cc -o in_filt in_filt.c -lexoIIv2c -lnetcdf -lnsl -lm
```

7. Now the optimization can be run. To execute the optimization, issue the command:

```
dakota -i tut.in
```

Wait until the thing finishes and enjoy the results.

Dryer Design Example

This section presents an extension of the tutorial problem to an example problem that you care about. The shell program changes trivially, the input filter doesn't change at all, and only the cost function evaluation changes in the output filter. The problem that is being solved is the multilayer drying problem shown in Figure 2. The one dimensional problem has two solvents and a substrate. The cost function is the concentration of `solvent 0` at the end of the simulation, which for this case is 200 sec. The design variable is the oven temperature, which has a constraint of 370K to prevent boiling.

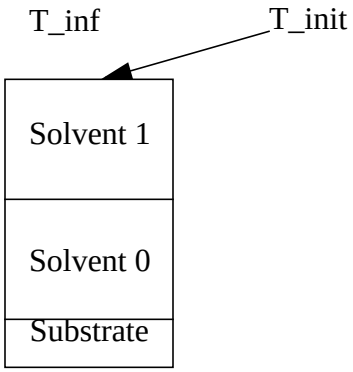


Figure 2 Drying Problem setup

Dryer Design Tutorial:

1. There are a few differences in the input specification to DAKOTA. The specification is in `dryer.in`. The first is the change in the name of the analysis driver:

```
analysis_driver=        'dryer.sh'
```

   The variable description also changes:

```
variables,                                          \
       continuous_design = 1                                  \
       cdv_initial_point       300.0 \
       cdv_upper_bounds     600.0  \
       cdv_lower_bounds       0.   \
       cdv_descriptor    'T_inf'
```

   The final change is in the responses section. Here the number of constraints must be changed to reflect the current problem:

```
num_nonlinear_constraints = 1           \
```

2. The shell function `dryer.sh` is identical to `cost.sh` described above except for the GOMA evaluation. To look at the file execute:

```
more dryer.sh
```

   The file looks like

```
#! /bin/csh -f
#
#This shell file evaluates the cost function
#for a dakota run
#
in_filt $argv[1] dryer.app

goma -a -i ml_input -se stderr -so stdout

dryer $argv[1] $argv[2]>>& goma.src
```

3. The input filter is identical. To check this execute:

```
more in_filt.c
```

   Note that the output of the input filter is the file `dryer.app` as can be seen from the file `dryer.sh`, which contains the oven temperature. This file is read by the GOMA input file `Defs.app`. Check this file now to see the include statement at the top of the file.

4. The simulation is identical to the template `dryer.ml` provided to you in your distribution. If you are not familiar with it, become familiar with it.

5. The major changes are in the output filter, `dryer.c` for this problem. Only the cost function evaluation will be discussed. The remaining code is as it was described above. Any variables in all capital letters are defined in the header.

   The cost function requires interrogating the EXODUS file that is generated by GOMA for the concentration of the solvent `Y0` at a node near the substrate (node 8). Open the file `dryer.c` by executing:

   ```
   emacs dryer.c
   ```

   Move the cursor down to the portion of the code where the function `output_filter()` is defined. After the definition of the necessary variables, the first line of code opens the EXODUS file using an EXODUS provided subroutine:

   ```
   /* page 25 of SAND92-2137 */
   /* Open file */

    CPU_word_size=sizeof(float);
    IO_word_size=0;

    exoid=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&version);
   ```

   Note that the comments give the reference page in the EXODUS users manual which is available on-line at `http://sass577.endo.sandia.gov/SEACAS/SEACAS.html`

   The `exoid` output is used to reference the file. It is of type `int`. Now the cost function `cost_fun()` is called with the argument being `exoid`. After some variable definitions, the number of variables are determined from the database and the array for the variable names is set up:

   ```
   /* page 133 of SAND92-2137 */

   error=ex_get_var_param(exoid,"n",&num_nodal_var);

   for (i=0;i<num_nodal_var;i++)
    var_names[i] = (char *) calloc((MAX_STR_LNG+1),sizeof(char));
   ```

   Next, the variable names are extracted and the index of the appropriate variable, `Y0`, is determined. The variables are referenced in the database by their index and this will be needed when extracting the concentration time history.

   ```
   /* page 137 of SAND92-2137 */
   error=ex_get_var_names(exoid,"n",num_nodal_var,var_names);

   /* find the velocity variables */

   for (i=0;i<num_nodal_var;i++)
    {
    if (strcmp(CONC,var_names[i])==0)  CONC_var_index=i+1;
    }
   ```

   Next, the number of time steps are determined and the arrays to hold all the time step values and the values of the concentration history at node 8 are allocated. The array times will contain the time axis.

   ```
   /* page 41 of SAND92-2137 */
   /* determine number of time steps and use the last one */
   error=ex_inquire(exoid,EX_INQ_TIME,&num_time_steps,&fdum,cdum);
   ```

```
concentration = (float *) calloc(num_time_steps,sizeof(float));
times = (float *) calloc(num_time_steps,sizeof(float));

/* page 143 of SAND92-2137 */
error = ex_get_all_times(exoid,times);
```

Now the concentration history at node 8 is read and the last value of the concentration is used for the cost function

```
/* page 167 of SAND92-2137*/

error = ex_get_nodal_var_time(exoid, CONC_var_index,NODE,1,

num_time_steps,concentration);
printf(" %g   %g \n",concentration[0],concentration[num_time_steps-1]);
J_cost=concentration[num_time_steps-1];
```

6. To compile a program with EXODUS subroutines in it, execute a command similar to:

```
cc -o in_filt in_filt.c -lexoIIv2c -lnetcdf -lnsl -lm
```

7. To run the simulation, just type:

```
dakota -i dryer.in
```

8. Sit back and watch it run.

Dryer Parameter Study in Fortran:

This section will go through an example of a FORTRAN interface between DAKOTA and GOMA. The example will be a multi-dimensional parameter study using the same cost function as the previous example, namely the concentration of the solvent at the substrate at the end after 200 sec. The two variables that will vary are the oven temperature, `T_inf`, and the convection coefficient, `Kh`.

1. The input specification, **`dryer.in`**, for DAKOTA is as follows:

```
interface,                                                          \
       application system,                                          \
         input_filter =          'NO_FILTER'   \
         output_filter =         'NO_FILTER'   \
         analysis_driver=        'dryer.sh'                        \
         parameters_file=        'params.in'\
         results_file=           'results.out'\
         analysis_usage =        'DEFAULT'


variables,                                                          \
       continuous_design = 2                                        \
         cdv_initial_point       300.0   -50 \
         cdv_upper_bounds        600.0   -50\
         cdv_lower_bounds        0.     0\
         cdv_descriptor    'T_inf'      'Kh'


responses,                                                          \
       num_objective_functions = 1                                  \
       num_linear_constraints = 0                                   \
       num_nonlinear_constraints = 1                                \
       no_gradients                                                 \
       no_hessians

strategy,                                                           \
       single_method


method,                                          \
        multidim_parameter_study\
```

```
partitions = 10 10
```

The main difference between this file and the ones discussed in the previous examples is the `variables` section and the `method` section.

The next file necessary is the shell file `dryer.sh` which runs the simulation and controls the cost function evaluation. It is pretty simple and has been discussed in both the previous examples:

```
#! /bin/csh -f
#
#This shell file evaluates the cost function
#for a dakota run
#
in_filt

goma -a -i ml_input -se stderr -so stdout

dryer
```

2. The input filter in FORTRAN is a little less general than the one written in C. It is not easy to pass command line arguments in FORTRAN and therefore the files read and written by the input filter have to be hard coded. It is imperative that the files coded to be read in the input filter are identical to those used in the dakota input specification. This means that `file_tag` and `file_save` cannot be used, nor can the default file names for the parameter file or the results file.

```
      program in_filt
c
c     This is a poor man's version of
c     the c program in_filt.c
c     LEARN C!!!
c

      integer i, nparam, nfun
      real dparam
      character*10 tag, junk

       open(22,file='params.in',status='old')
       open(33,file='dryer.app',status='unknown')

       read(22,*) nparam, tag, nfun, junk

       do 10 i=1, nparam
         read(22,*) dparam,tag
         write(33,'(1x,a2,a10,a1,f16.8,a1)') "#{",tag,"=",dparam,"}"
   10  continue

       end
```

3. The function that evaluates the cost function, `dryer.f`, is now described. As with the input filter the filenames have to be hard coded, limiting the generality of the code. The main program does little except call the appropriate subroutines in the appropriate order

```
      program dryer
      include '/usr/local/inc/exodusII.inc'
      character*12 infile, outfile
      integer asv(3), nparam, ng
      real params(2)

      infile = 'params.in'
      outfile = 'results.out'

      call asvrd(infile, nparam, ng, asv, params)
```

```
      call outfilt(asv, nparam, params, ng, outfile)

      stop
      end
```

The first subroutine, `asvrd()`, reads the parameters file, `params.in`, and determines the values of the parameters and the ASV

```
      subroutine asvrd(infile, nparam, ng, asv, params)
      character*12 infile
      character*50 junk, junk1
      integer i, nparam, ng, asv(3)
      real params(2)

      open(unit=22, file=infile, status='old')

      read(22,*) nparam, junk, ng, junk1
      ng=ng-1

      do 10 i=1,nparam
        read(22,*) params(i), junk
10    continue

      do 20 i=1,ng+1
        read(22,*) asv(i), junk
        write(*,*) asv(i), junk
20    continue

      close(22)
      end
```

The next subroutine, `outfilt()`, opens the EXODUS database and controls the writing of the file, `results.out` for DAKOTA to read. It does a lot of checks to make sure that the function values and their gradients that DAKOTA asks for through the ASV are, in fact, available

```
      subroutine outfilt(asv, nparam, params, ng, outfile)
      include '/usr/local/inc/exodusII.inc'
      integer asv(3), i, nparam, ng
      real params(2)
      character*12 outfile
      real J_cost, g(2)

      integer cpu_ws, exopen, exread, io_ws, idexo, ierr
      real vers

      cpu_ws=0
      io_ws=0

c page 25 of SAND92-2137

      idexo = EXOPEN ("out.exoII", EXREAD, cpu_ws, io_ws, vers, ierr)

      g(1) = params(1) - 370.0

      J_cost = costf(idexo)

      open(unit=33, file=outfile, status='unknown')

      if (asv(1) .gt. 3) then
        write(*,*) 'Hessian is not available, exiting '
        call exit(0)
      endif

      if (asv(1) .gt. 2) then
        write(*,*) 'Gradient is not available, exiting '
        call exit(0)
```

```
      endif

      if (asv(1) .eq. 1 .or. asv(1) .eq. 3 .or. asv(1) .eq. 5 ) then
        write(33,*) J_cost, ' f'
      endif

      do 30 i=1,ng

        if (asv(i) .gt. 3) then
          write(*,*) 'Hessian is not available, exiting '
          call exit(0)
        endif

        if (asv(i) .gt. 2) then
          write(*,*) 'Gradient is not available, exiting '
          call exit(0)
        endif

        if (asv(i) .eq. 1 .or. asv(i) .eq. 3 .or. asv(i) .eq. 5 ) then
          write(33,*) g(i), ' g1'
          write(*,*) g(i)
        endif

 30   continue

      end
```

The last function, `costf()`, calculates determines what the value of the solvent at the substrate is at the end of the simulation (200 sec). It uses a lot of calls from the EXODUS subroutine library and page numbers in the EXODUS reference guide are give to facilitate reading the code. The variable exoid is used to reference the EXODUS database file that the GOMA results will be read from.

```
      real function costf(idexo)
      include '/usr/local/inc/exodusII.inc'
      integer cvarind, extims, i, idexo, ntime, nvar, ierr

      real redum, time
      real time(500), concen(500)
      character*(MXSTLN) vname(20)
      character cdum
```

First, we need to know how many variables are in the database

```
c page 133 SAND92-2137

      call EXGVP(idexo, "n", nvar, ierr)

c page 137 SAND92-2137
```

Next, we read the variable's names in and determine which one is the one of interest. In this case we are interested in `Y0`.

```
      do 40 i=1,nvar
        if (vname(i) .eq. "Y0") then
          cvarind = i
        endif
 40   continue
```

Now we find out how many time steps are in the database

```
c page 41 of SAND92-2137

      call EXINQ(idexo, EXTIMS, ntime, redum, cdum, ierr)
c page 144 of SAND92-2137
```

```
      call EXGATM(idexo, time, ierr)
```

Finally we read in all the values of `Y0` through time and take the last one, then close the file

```
c page 167 of SAND92-2137

      call EXGNVT(idexo, cvarind, 8, 1, ntime, concen, ierr)

      costf=concen(ntime)

c page 27 of SAND92-2137

      call EXCLOS(idexo,ierr)
      return
      end
```

4. To compile a FORTRAN program with EXODUS commands in it, execute a command similar to:

```
f77 -o dryer dryer.f -lexoIIv2for -lexoIIv2c -lnetcdf -lnsl
```

5. To run the simulation, just type

```
dakota -i dryer.in
```

6. Sit back and watch it run.

Slot Coater Example

The slot coater example utilizes the failure capture option in DAKOTA. There are two ways to insure a solution throughout the optimization: The first is to set the relaxation schedule very conservatively and the other is to rely on continuation. By relying on continuation, the optimization runs significantly faster. In this optimization, there was no relaxation used.

The parameterization used for this example is shown in Figure 3. Only the gap and the angle a were used in the optimization. The parameters used for the starting point of the optimization were taken from

Sator (1990), *Slot Coating*, PhD. Thesis University of Minnesota, Available on University Microfilms, Ann Arbor, Michigan.

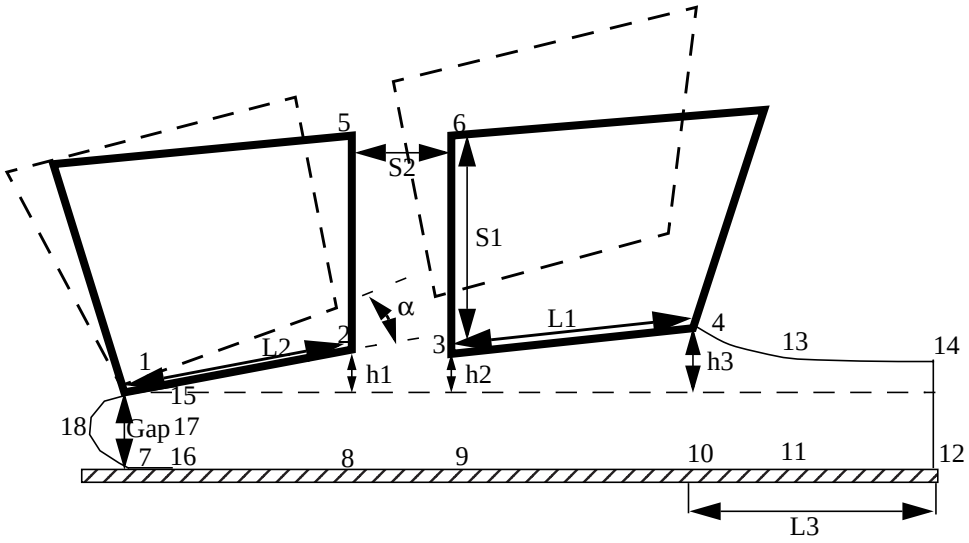


Figure 3 Slot coat parameterization

The cost function used for this optimization is

$$J = \left(\frac{28.5}{0.02}\right)\frac{\partial}{\partial u_{web}}(x_{dcl}) + \left(\frac{2850}{0.01}\right)\frac{\partial}{\partial P_{vac}}(x_{dcl}) \qquad (31)$$
$$0.35 < Gap < 0.7$$
$$-0.2 < \alpha < 0.2$$

The design variable, as it is currently set up, is the gap length and the angle α. The cost function was choosen to minimize the sensitivity of the movement of the dynamic contact line to changes in the webspeed or the back pressure.

Slot Coater Tutorial:

1. As with all the examples before, the first file necessary is the DAKOTA input specification. This example is identical to all the optimization problems so far except for the specification of the design variables and the addition of the failure capture command in the interface specification. The file is called `slot.in`. The changed portions of the specification for this problem is

```
interface,                                                  \
 application system,                                         \
 input_filter =          'NO_FILTER'           \
 output_filter =         'NO_FILTER'           \
 analysis_driver=        'slot.sh'             \
 parameters_file=        'params.in'   \
 results_file=           'results.out' \
 analysis_usage =        'DEFAULT'     \
 failure_capture continuation

######
```

```
variables,                                          \
 continuous_design = 2                               \
 cdv_initial_point           0.05  0.0\
 cdv_upper_bounds           0.07     0.2\
 cdv_lower_bounds          0.035     -0.2\
 cdv_descriptor      'Gap_new'  'alpha_new'
```

2. The C-shell file, `slot.sh`, should look pretty familiar also

```
#! /bin/csh -f
#
#  This shell file evaluates the cost function
#    for a dakota run
#
in_filt $argv[1] slot.app

goma -a -i slot_input -se stderr -so stdout

slot $argv[1] $argv[2]
```

3. The input filter `in_filt.c` is identical to all the previous input filters.

4. The major difference is in the cost function, `slot.c`. The subroutines `main()` and `asv_read()` are the same. However, the routine `output_filter()` has been changed to incorporate a failure capturing scheme. There has been added to GOMA four global variables that indicate the convergence status of the GOMA simulation. They are:

   - CONVBoolean convergence (1=> converged, 0=> not converged)
   - NEWT_ITNumber of Newton Iterations specified in the GOMA input file
   - MAX_ITNumber of Newton Iterations taken by the simulation
   - CONVRATEThe log10 relative convergence rate at the second to last and the last iteration taken

   The subroutine converge in `slot.c` takes care of reading these values. The subroutine `out_filt` look like

```
void output_filter(int *asv, int n_param, double *params, int n_g,
          FILE *output_file)
{
 char filename[]=GOMA_FILE;
 int CPU_word_size, IO_word_size;
 float version;
 int exoid, i;
 double J_cost;
 double *g;
 int newt_it, max_it,error;
 double convrate;

 g=(double *)calloc(n_g ,sizeof(double));

/* page 25 of SAND92-2137 */
/* Open file */

 CPU_word_size=sizeof(double);
 IO_word_size=0;
```

   This section of the code is the most different. Note how the EXODUS file is opened, then the convergence is checked. If the simulation didn't converge, a failure is flagged and the program exits. If the simulation didn't converge but it ran out of newton iterations, then the

program exits and a “1” is returned so the shell program can rerun GOMA (not yet implemented). If it has converged, then it writes the results.out file as before.

```
exoid=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&version);

error = converge(exoid, &max_it, &newt_it, &convrate);

if (!error) {
  /* determine cost function and constraints*/

  system("cp soln.dat contin.dat");

  g[0] = - 0.5e-4;
  J_cost =  cost_fun(exoid);

  J_cost=J_cost*J_cost;

  printf("J= %g\n",J_cost);

  /* write dakota output file */

  if (asv[0]>3) {
    printf("Hessian is not available, exiting\n");
    exit(-1);
    }

  if (asv[0]>2) {
    printf("Gradient is not available, exiting\n");
    exit(-1);
    }

  if (asv[0]==1 || asv[0]==3 ||asv[0]==5)  fprintf(output_file,
     "%g f\n",J_cost);

  for (i=1;i<=n_g;i++) {
    if (asv[i]==1 || asv[i]==3 || asv[i]==5) {
      fprintf(output_file,"%g c%d\n",g[i-1],i);
      }
else {
      printf("Number of parameters is probably wrong:      exiting.\n");
      exit(1);
      }
    }
  return;
  }
if (newt_it == max_it && convrate > 0.0) {
  printf("Not converged!! \n");
  exit(1);
  }
else {
  fprintf(output_file,"FAIL\n");
  }

free(g);
}
```

5. The `converge()` routine is fairly basic. It reads the global variables from the EXODUS database, then sends them back.

```
int converge(int exoid, int *max_it, int *newt_it, double *convrate)
{
  int i, inewt, iconv, imax, irate;
  int ret_int, ntime, nvar, conv;
  int error;
  char *cdum=0, *gvar_name[NUM_G_VAR];
  float fdum;
  double gvar[NUM_G_VAR];

  error=ex_inquire(exoid, EX_INQ_TIME, &ntime, &fdum, cdum);
```

```
  error=ex_get_var_param(exoid, "g", &nvar);

  for (i=0; i<nvar;i++) gvar_name[i]= (char *) calloc((MAX_LINE+1),
    sizeof(char));

  error=ex_get_var_names(exoid, "g",nvar, gvar_name);

  for (i=0;i<nvar;i++) {
    if (strcmp(CON_VAR,gvar_name[i])==0) iconv=i;
    if (strcmp(NEWT_VAR,gvar_name[i])==0) inewt=i;
    if (strcmp(MAX_VAR,gvar_name[i])==0)  imax=i;
    if (strcmp(RATE_VAR,gvar_name[i])==0) irate=i;

    }

  /* Page 159 SAND92-2137 */
  error=ex_get_glob_vars(exoid, ntime, nvar, gvar);

  if (error == 0) {
    *newt_it=(int) gvar[inewt];
    *max_it=(int) gvar[imax];
    *convrate= gvar[irate];
    conv=(int) gvar[iconv];
    }
  else {
    *newt_it= -1;
    *max_it= -1;
    *convrate= -999999.0;
    conv=0;
    }
  return !conv;
}
```

The cost function evaluation subroutine, `cost_fun()`, is more complicated. Actually it isn't that difficult, it just looks that way. Basically there are two files, `webspeed.app` and `vacuum.app` which are read by `cost_fun()`. First, the nominal position of the dynamic contact point is read. Procedure `cost_fun()` then perturbs the values in `webspeed.app` and calls GOMA, then reads the perturbed value of the dynamic contact point. This is repeated for the back pressure. The perturbed values are then used for a finite difference calculation.

```
double cost_fun(int exoid_nom)
{
 int i, CPU_word_size, IO_word_size;
 int error, exoid_delta ,idum;
 int ns_num_nodes, *ns_node_list;
 double fdum, J1, J2;
 float version;
 double webspeed_nom,webspeed_delt;
 double Pvacuum_nom,Pvacuum_delt, g1,g2;
 double *ns_X,*ns_Y,*ns_Z,*ns_displx_nom, *ns_displx_delt;
 char filename[]=GOMA_FILE,cdum[9];
 FILE *in_file;

 error=ex_get_node_set_param(exoid_nom, NSET, &ns_num_nodes,&idum);
 ns_node_list=(int *) calloc(ns_num_nodes,sizeof(int));

 error=ex_get_node_set(exoid_nom,NSET, ns_node_list);

 ns_X=(double *) calloc(ns_num_nodes,sizeof(double));
 ns_Y=(double *) calloc(ns_num_nodes,sizeof(double));
 ns_Z=(double *) calloc(ns_num_nodes,sizeof(double));
 ns_displx_nom=(double *) calloc(ns_num_nodes,sizeof(double));
 ns_displx_delt=(double *) calloc(ns_num_nodes,sizeof(double));

 get_displ(exoid_nom,NSET,ns_num_nodes,ns_node_list, ns_X, ns_Y, ns_Z,
```

```
            ns_displx_nom);

          /******************/
          in_file=fopen(WEBFILE,"r");
          fscanf(in_file,"${%s = %lf}",cdum,&webspeed_nom);

          fclose(in_file);

          webspeed_delt=(1.0+FDEPS)*webspeed_nom;
          in_file=fopen(WEBFILE,"w");
          fprintf(in_file,"${webspeed = %f}\n",webspeed_delt);
          fclose(in_file);

          system("/home/prschun/.sun5/bin/goma -a -i slot_input -se
          stderr -so stdout");
          /*system("/home/prschun/.sun5/bin/goma -a -i slot_input");*/
          in_file=fopen(WEBFILE,"w");
          fprintf(in_file,"${webspeed = %f}\n",webspeed_nom);
          fclose(in_file);

          CPU_word_size=sizeof(double);
          IO_word_size=0;


    exoid_delta=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&version);

          get_displ(exoid_delta,NSET,ns_num_nodes,ns_node_list, ns_X, ns_Y, ns_Z,
            ns_displx_delt);
          /***************/
          g1= webspeed_nom/Ls;g1=1.0e3;
          J1= (ns_displx_delt[0] - ns_displx_nom[0])/(webspeed_delt-webspeed_nom);
          /******************/
          in_file=fopen(PRESSFILE,"r");
          fscanf(in_file,"${%s = %lf}",cdum,&Pvacuum_nom);
          fclose(in_file);
          Pvacuum_delt=(1.0+FDEPS)*Pvacuum_nom;
          in_file=fopen(PRESSFILE,"w");
          fprintf(in_file,"${vacuum = %f}\n",Pvacuum_delt);

          fclose(in_file);
          system("/home/prschun/.sun5/bin/goma -a -i slot_input -se stderr -so
          stdout");
          /*system("/home/prschun/.sun5/bin/goma -a -i slot_input");*/
          in_file=fopen(PRESSFILE,"w");
          fprintf(in_file,"${vacuum = %f}\n",Pvacuum_nom);
          fclose(in_file);

          CPU_word_size=sizeof(double);
          IO_word_size=0;


    exoid_delta=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&version);

          get_displ(exoid_delta,NSET,ns_num_nodes,ns_node_list, ns_X, ns_Y, ns_Z,
            ns_displx_delt);

          /***************/
          g2=abs(Pvacuum_nom/Ls);g2=1.0e7;
          J2= (ns_displx_delt[0] - ns_displx_nom[0])/(Pvacuum_delt - Pvacuum_nom);
          /*printf("J1 = %e , J2 = %e \n",J1,J2);*/
          return ALPHA*J1 + BETA*J2;
        }
```

6. Now compile the code and run DAKOTA.

Appendix

This appendix will briefly describe the process of using DAKOTA with another analysis driver such as FIDAP. The procedure is basically identical to when GOMA is used for the analysis.

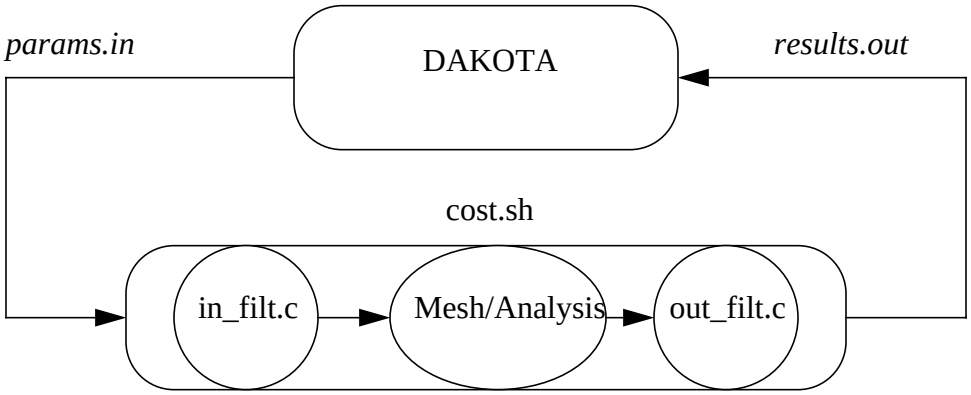


Figure A. DAKOTA interface scheme

1. Set up optimization by writing a DAKOTA input file. (See page 2-4 for example)
2. Write an input filter to take the file params.in generated by DAKOTA (format of parameter file is on page 8) and write and output file that can be used by your analysis code. An easy way to do this is to use APREPRO. If APREPRO is always used, the input filter in_filt.c can be written generally enough so that it can be used for all optimizations. (see pages 11-13)
3. Now parameterize your model so that the design variables that you want to vary can be easily changed by APREPRO. Make sure the output from your code has the information you will need to evaluate your cost function.
4. Write a program (out_filt.c) that takes the output from your code, evaluates your cost function, and writes a file (results.out) that (i) has the information requested from DAKOTA (this is specified in params.in) and (ii) is in a format that DAKOTA can read. (see pages 13 - 16)
5. In this tutorial, the programs that result from steps 2-4 are driven by a shell program cost.sh. DAKOTA, therefore, only has to call the shell program to evaluate the cost function.

```
Copy to:
MS0826 9111        Dayfile
MS 08269111        W. L. Hermina
MS 08269111        P. R. Schunk
MS0826 9111        R. R. Rao
MS0826 9111   P. A. Sackinger
MS 08349112   T. A. Baer
MS 08269111   D. A. Labreche
MS 05579741   T. W. Simmermacher

Dr. Richard A. Cairncross
Drexel University
Department of Chemical Engineering
Philadelphia, PA  19104
```

Dr. Ian Gates
University of Minnesota
Department of Chemical Engineering and Materials Science
421 Washington Ave. SE
Minneapolis, MN 55455

Additional References

Refer to

- [Eldred, M.S., Hart, W.E., Bohnhoff, W.J., Romero, V.J., Hutchinson, S.A., and Salinger, A.G., 1996]
- [Eldred, M.S., Outka, D.E., Bohnhoff, W.J., Witkowski, W.R., Romero, V.J., Ponslet, E.R., and Chen, K.S., 1996]

for procedures and lessons learned in interfacing with complex engineering simulation codes. Key findings in complex engineering applications are also summarized in [Eldred, M.S., 1998].

Anderson, G., and Anderson, P., 1986 *The UNIX C Shell Field Guide*, Prentice-Hall, Englewood Cliffs, NJ.

Byrd, R.H., Schnabel, R.B., and Schultz, G.A., 1988 *Parallel quasi-Newton Methods for Unconstrained Optimization*, Mathematical Programming, 42(1988), pp. 273-306.

Coplien, J.O., 1992 *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA.

Dennis, J.E., and Torczon, V.J., 1994 *Derivative-Free Pattern Search Methods for Multidisciplinary Design Problems*, paper AIAA-94-4349 in Proceedings of the 5th AIAA/ USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept. 7-9, 1994, pp. 922-932.

Eckstein, J., Hart, W.E., and Phillips, C.A., 1997 *Resource management in a parallel mixed integer programming package*, Proceedings of the 1997 Intel Supercomputer Users Group Conference (http://www.cs.sandia.gov/ISUG97/program.html), Albuquerque, NM, June 11-13, 1997.

Eldred, M.S., and Schimel, B.D., 1999 *Extended Parallelism Models for Optimization on Massively Parallel Computers*, paper 16-POM-2 in Proceedings of the 3rd World Congress of Structural and Multidisciplinary Optimization (WCSMO-3), Amherst, NY, May 17-21, 1999.

Eldred, M.S., and Hart, W.E., 1998 *Design and Implementation of Multilevel Parallel Optimization on the Intel TeraFLOPS*, paper AIAA-98-4707 in Proceedings of the 7th AIAA/ USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, Sept. 2-4, 1998, pp. 44-54.

Eldred, M.S., 1998 *Optimization Strategies for Complex Engineering Applications*, Sandia Technical Report SAND98-0340, Sandia National Laboratories, Albuquerque, NM.

Eldred, M.S., Hart, W.E., Bohnhoff, W.J., Romero, V.J., Hutchinson, S.A., and Salinger, A.G., 1996 *Utilizing Object-Oriented Design to Build Advanced Optimization Strategies with Generic Implementation*, paper AIAA-96-4164 in Proceedings of the 6th AIAA/USAF/NASA/ ISSMO Symposium on Multidisciplinary Analysis and Optimization, Bellevue, WA, Sept. 4-6, 1996, pp. 1568-1582.

Eldred, M.S., Outka, D.E., Bohnhoff, W.J., Witkowski, W.R., Romero, V.J., Ponslet, E.R., and Chen, K.S., 1996 *Optimization of Complex Mechanics Simulations with Object-Oriented Software Design*, Computer Modeling and Simulation in Engineering, Vol. 1, No. 3, August 1996. Revised and extended from Eldred, M.S., Outka, D.E., Fulcher, C.W., and Bohnhoff, W.J., *Optimization of Complex Mechanics Simulations with Object-Oriented Software Design*, paper AIAA-95-1433 in Proceedings of the 36th AIAA/ASME/ ASCE/AHE/ASC Structures, Structural Dynamics, and Materials Conference, New Orleans, LA, April 10-13, 1995, pp. 2406-2415.

Friedman, J. H., 1991 *Multivariate Adaptive Regression Splines*, Annals of Statistics, Vol. 19, No. 1, March 1991, pp. 1-141.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H., 1986 *User's Guide for NPSOL (Version 4.0): A Fortran Package for Nonlinear Programming*, System Optimization Laboratory, TR SOL-86-2, Stanford University, Stanford, CA.

Gill, P.E., Murray, W., and Wright, M.H., 1981 *Practical Optimization*, Academic Press, San Diego, CA.

Gropp, W., and Lusk, E., 1996 *User's Guide for mpich, a Portable Implementation of MPI*, Argonne National Laboratory, Mathematics and Computer Science Division, Report ANL/MCS-TM-ANL-96/6.

Gropp, W., Lusk, E., and Skjellum, A., 1994 *Using MPI, Portable Parallel Programing with the Message-Passing Interface*, The MIT Press, Cambridge, MA.

Hart, W.E., 1997 *SGOPT, A C++ Library of (Stochastic) Global Optimization Algorithms*, Sandia Report SAND98-xxxx, Sandia National Laboratories, Albuquerque, NM.

Kernighan, B.W., and Ritchie, D.M., 1988 *The C Programming Language*, Second Edition, Prentice Hall PTR, Englewood Cliffs, NJ.

Meza, J.C., 1994 *OPT++: An Object-Oriented Class Library for Nonlinear Optimization*, Sandia Report SAND94-8225, Sandia National Laboratories, Livermore, CA.

Meza, J.C., and Plantenga, T.D., 1995 *Optimal Control of a CVD Reactor for Prescribed Temperature Behavior*, Sandia Technical Report SAND95-8224, Sandia National Laboratories, Livermore, CA.

Moen, C.D., Spence, P.A., and Meza, J.C., 1995 *Optimal Heat Transfer Design of Chemical Vapor Deposition Reactors*, Sandia Technical Report SAND95-8223, Sandia National Laboratories, Livermore, CA.

Moen, C.D., Spence, P.A., Meza, J.C., and Plantenga, T.D., 1996 "Automatic Differentiation for Gradient-Based Optimization of Radiatively Heated Microelectronics Manufacturing Equipment", paper AIAA-96-4118 in *Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Bellevue, WA, pp. 1167-1175.

Ponslet, E.R., and Eldred, M.S., 1996 “Discrete Optimization of Isolator Locations for Vibration Isolation Systems: an Analytical and Experimental Investigation,” paper AIAA-96-4178 in *Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Bellevue, WA, Sept. 4-6, 1996, pp. 1703-1716. Also appears as Sandia Technical Report SAND96-1169, May 1996.

Schunk, P.R., Sackinger, P.A., Rao, R.R., Chen, K.S., Cairncross, R.A., 1995 *GOMA - A Full-Newton Finite Element Program for Free and Moving Boundary Problems with Coupled Fluid/ Solid Momentum, Energy, Mass, and Chemical Species Transport: User’s Guide*, Sandia Report SAND95-2937, Sandia National Laboratories, Albuquerque, NM.

Sjaardema, G.D., 1992 *APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses*, Sandia Report SAND92-2291, Sandia National Laboratories, Albuquerque, NM.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J., 1996 *MPI: The Complete Reference*, MIT Press, Cambridge, MA.

Tong, C.H., and Meza, J.C., 1997 *DOOMSDACE: A Distributed Object-Oriented Software with Multiple Samplings for the Design and Analysis of Computer Experiments*, Sandia Technical Report SAND97-XXXX (draft as yet unpublished).

Vanderplaats Research and Development, 1995 *DOT Users Manual, Version 4.20*, Inc., Colorado Springs.

Weatherby, J.R., Schutt, J.A., Peery, J.S., and Hogan, R.E., 1996 *Delta: An Object-Oriented Finite Element Code Architecture for Massively Parallel Computers*, SAND96-0473.

Zimmerman, D.C., 1996 *Genetic Algorithms for Navigating Expensive and Complex Design Spaces*, Final Report for Sandia National Laboratories contract AO-7736 CA 02, Sept. 1996.